

Gabriele Bellomia

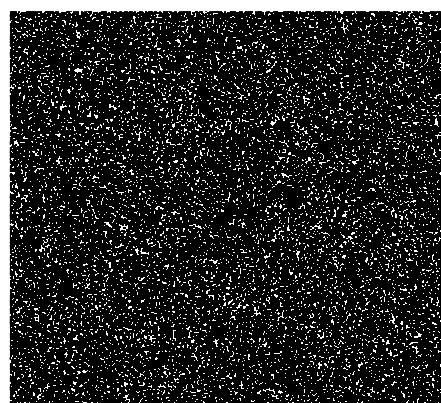
Analisi Statistica dei Dati
- Esercizi Svolti -



A.A. 2016/2017

Analisi Statistica dei Dati
- Esercizi Svolti -

Gabriele Bellomia
Matricola 821904



A.A. 2016/2017

Indice

Introduzione e nota sui linguaggi di programmazione utilizzati	3
Esercizio 1	4
Esercizio 2	14
Esercizio 3	18
Esercizio 4	35
Esercizio 5	42
Esercizio 6	63
Esercizio 7	73
Lista dei codici	79

Introduzione e nota sui linguaggi di programmazione utilizzati

La seguente raccolta di esercizi svolti costituisce buona parte della prova d'esame per il corso di *Analisi Statistica dei Dati* presso il Dipartimento di Fisica dell'Università degli Studi di Milano-Bicocca. I sette esercizi assegnati spaziano grosso modo tra tutti gli argomenti affrontati durante tale corso semestrale e rappresentano esempi applicativi delle tecniche illustrate a lezione. Gli ambiti in cui tali applicazioni si inseriscono sono piuttosto variegati, da esercizi molto formali e astratti, quali la ricerca di minimi globali di celebri funzioni "patologiche" o il calcolo di integrali per mezzo di tecniche Montecarlo, a vere e proprie analisi dati su esperimenti (simulati) di ottica e fisica delle particelle. Non manca naturalmente una buona palestra su tutti quei fondamenti di matematica numerica e scienza dei calcolatori che costituiscono il necessario precedente (tecno)logico per qualsiasi moderna pratica di analisi statistica dei dati.

La natura intrinsecamente multidisciplinare della materia trattata nonché la marcata disomogeneità che caratterizza i differenti percorsi di una laurea magistrale in fisica fanno sì che la modalità di svolgimento degli esercizi sia sostanzialmente molto libera nella scelta degli approcci e degli strumenti di cui avvalersi. Personalmente ho dunque scelto di utilizzare per lo svolgimento di tutti gli esercizi dei linguaggi di programmazione *interpretati* e con una sintassi di *alto livello*. Tale scelta è chiaramente mirata alla flessibilità e all'immediatezza di scrittura che a mio parere un'attività di lavoro così eterogenea intrinsecamente richiede. In particolare ho implementato quasi tutto in **Python**, linguaggio piuttosto moderno ma ormai caro agli ambienti della ricerca scientifica, con il vezzo di qualche incursione nel mondo di **R**, habitat ideale per qualunque applicazione di statistica e *data science*. Maggiori informazioni possono essere facilmente reperite nei rispettivi siti ufficiali: www.python.org e www.r-project.org, fermo restando che la documentazione specifica di eventuali funzioni e pacchetti particolari sarà citata nel corpo della presentazione, sotto forma di nota a piè di pagina (con link). Disseminati lungo il testo saranno anche i listati di tutti gli *script* implementati, con specificato il linguaggio e evidenziata opportunamente la relativa sintassi. L'indice dei codici listati può essere consultato a pagina 79.

Trieste, 21 gennaio 2019

Esercizio 1

- A. Si costruisca un generatore di numeri casuali distribuiti secondo una densità di Breit-Wigner.
- B. Costruito un generatore di numeri casuali uniforme tra 0 e 10 si discuta dei possibili test di casualità utilizzabili per le sequenze prodotte.

* * *

Per generare numeri distribuiti secondo la densità di probabilità di Breit-Wigner possiamo utilizzare il *metodo della cumulante*: detta $f(x)$ la distribuzione di interesse e detta $F(x)$ la sua funzione cumulante¹, il problema viene ricondotto alla generazione di numeri casuali *uniformi* nell'intervallo $[0, 1]$, che risultano essere in corrispondenza biunivoca con la sequenza desiderata attraverso la funzione inversa della cumulante.

Ciò è garantito dal seguente risultato generale: la variabile casuale $\xi = F(x)$ è distribuita secondo una densità di probabilità $g(\xi)$ data da

$$\begin{aligned} g(\xi) &= f(x) \cdot \left| \frac{dx}{d\xi} \right| \\ &= f(x) \cdot \left| \frac{d\xi}{dx} \right|^{-1} \\ &= f(x) \cdot \left| \frac{d}{dx} \int_{-\infty}^x f(x') dx' \right|^{-1} \\ &= f(x) \cdot |f(x)|^{-1} \\ &= 1. \end{aligned}$$

Dal momento che, per definizione di cumulante, si ha $\xi \in [0, 1]$ risulta evidente che ξ ha distribuzione uniforme nell'intervallo $[0, 1]$, per cui possiamo ottenere delle x distribuite secondo la densità $f(x)$ dalla relazione:

$$x = F^{-1}(\xi),$$

con ξ numeri casuali opportunamente generati per avere distribuzione uniforme in $[0, 1]$.

Nel nostro caso la procedura è piuttosto semplice dal momento che la distribuzione di Breit-Wigner ha cumulante calcolabile analiticamente:

$$f_{\text{BW}}(x) = \frac{\Gamma}{\pi(\Gamma^2 + (x - x_0)^2)}$$

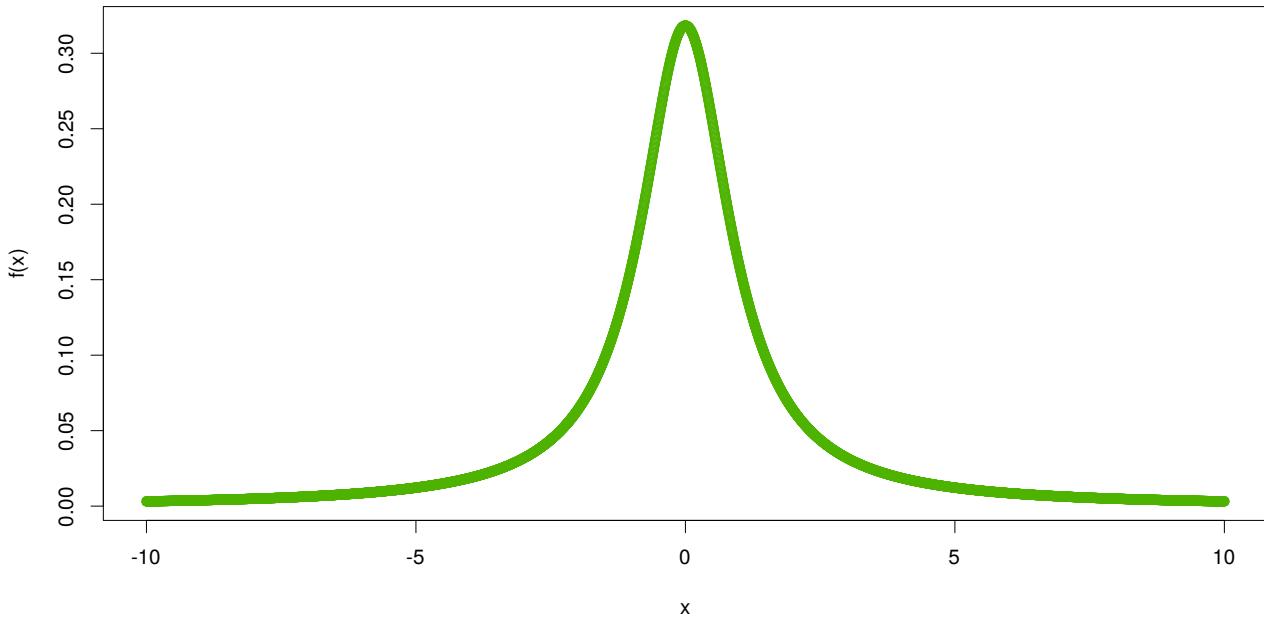
$$F_{\text{BW}}(x) = \frac{1}{\pi} \arctan \left[\frac{x - x_0}{\Gamma} \right] + \frac{1}{2}$$

con x_0 valore corrispondente al picco centrale della distribuzione e Γ sua larghezza HWHM (Cfr. Figura 1).

¹Ossia $F(x)$ sia definita come:

$$F(x) = \int_{-\infty}^x f(x') dx'$$

Figura 1: Distribuzione di Breit-Wigner per $\Gamma = 1$ e $x_0 = 0$.



Pertanto invertendo la relazione

$$\xi = \frac{1}{\pi} \arctan \left[\frac{x - x_0}{\Gamma} \right] + \frac{1}{2}$$

otteniamo l'equazione desiderata:

$$x = x_0 + \Gamma \tan \left[\pi \left(\xi - \frac{1}{2} \right) \right]$$

Per generare i numeri casuali ξ si è utilizzato il pacchetto `RNG` dell'ambiente statistico `R`, che permette di ottenere sequenze pseudo-random con periodo pari a $(2^{19937} - 1)$, per mezzo di un generatore di tipo "Twisted GFSR".²

In Figura 2 è rappresentato l'istogramma relativo a $N = 10^5$ estrazioni di x , raggruppate in 50 bins, con parametri Breit-Wigner $x_0 = 0$ e $\Gamma = 1$. L'ottima sovrapposizione con il grafico analitico della $f_{\text{BW}}(x)$ conferma la bontà del metodo implementato.³

Di seguito viene riportato il listato del codice `R` utilizzato:

```

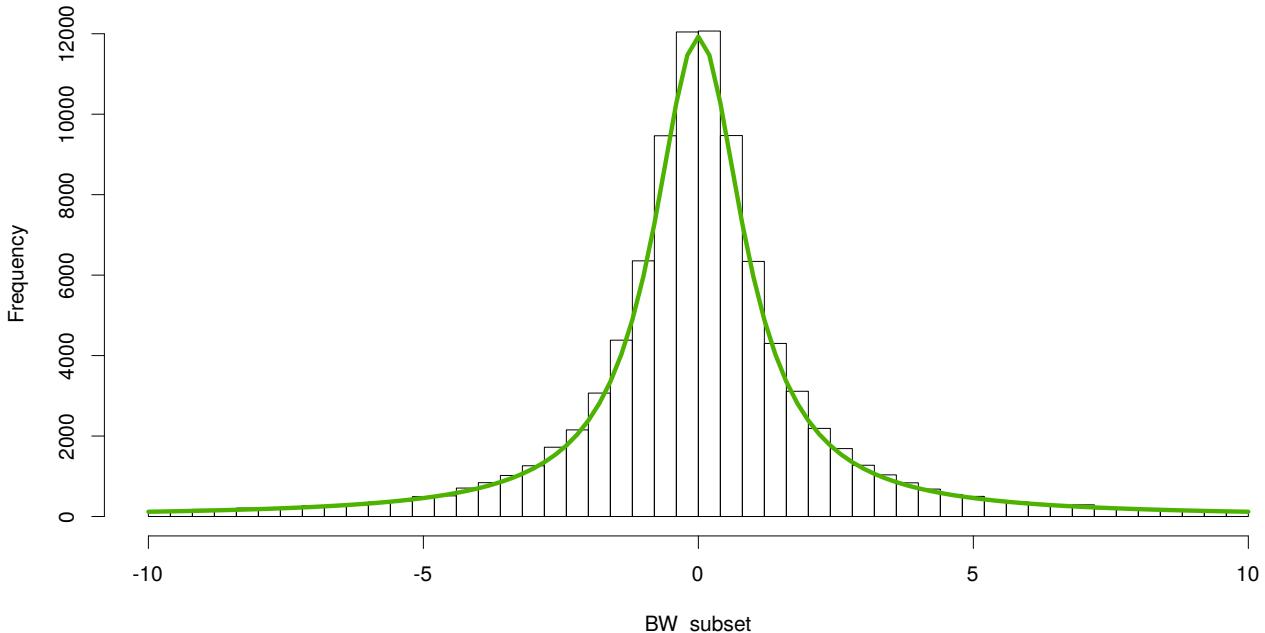
1 # Entering here the desired Breit-Wigner parameters:
2 x0 = 0
3 HWHM = 1
4
5 # Setting here an appropriate x domain (as a linear spaced vector)
6 x = seq(-10, 10, by = 0.04) # Note that "by" fixes the number of bins
7
```

²In particolare il pacchetto permette di eseguire svariati algoritmi per la generazione delle sequenze pseudo-random, selezionati specificando il parametro `rng.kind` nel codice. L'opzione di default implementa l'algoritmo *Marsenne-Twister*, sviluppato nel 1998 da Matsumoto e Nishimura [ACM article: [doi:10.1145/272991.272995](https://doi.org/10.1145/272991.272995)].

Per maggiori dettagli sul pacchetto `RNG`: <https://stat.ethz.ch/R-manual/R-devel/library/base/html/Random.html>

³Per quanto riguarda i limiti di affidabilità dell'algoritmo proposto si tenga presente che il numero di estrazioni N non dovrà mai essere comparabile con il periodo di ripetizione del generatore pseudo-random adoperato.

Figura 2: Uno degli istogrammi generati a confronto con il grafico analitico della distribuzione.



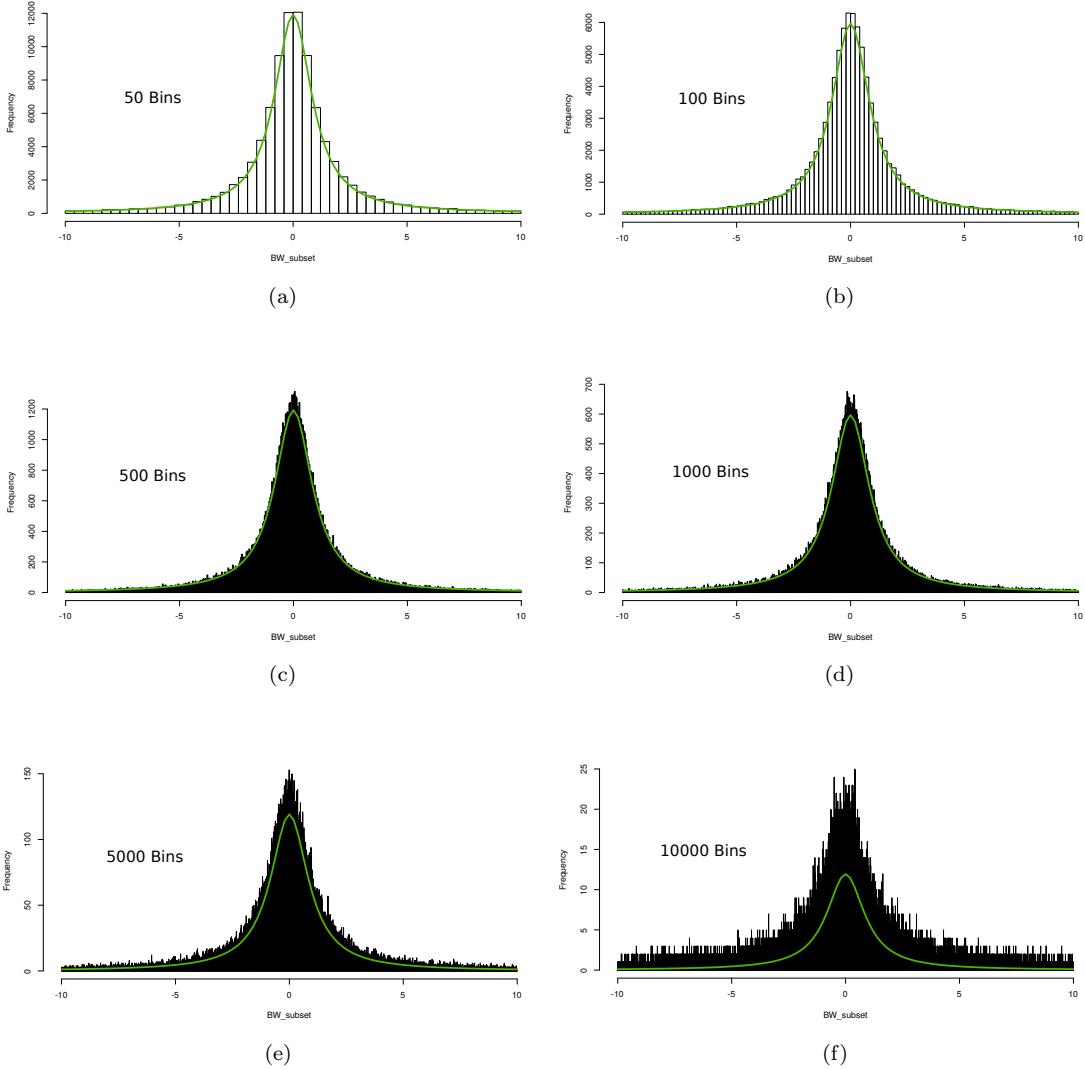
```

8 # Defining the Breit-Wigner function
9 f_x = HWHM / (pi*HWHM^2 + pi*(x-x0)^2)
10
11 # Plotting and saving a graph of the function
12 cairo_pdf('Breit-Wigner.pdf', width = 16, height = 9, pointsize = 18)
13 mycolor = rgb(0.3, 0.7, 0)
14 plot(x, f_x, xlab='x', ylab='f(x)', col=mycolor)
15 dev.off()
16
17 # Extracting the required N uniform random numbers (N is a parameter of choice)
18 N = 10^5
19 uniform_set = runif(N, 0, 1)
20
21 # Using inverse-cumulant equation to get Breit-Wigner dataset
22 BW_set = x0 + HWHM*tan(pi*(uniform_set-1/2))
23
24 # Defining histogram range and bins from x vector
25 hist_range = range(x)
26 bins = x
27
28 # Retrieving a proper subset of Breit-Wigner data
29 BW_subset = subset(BW_set, BW_set <= max(hist_range) & BW_set >= min(hist_range))
30
31 # Plotting and saving the histogram of Breit-Wigner dataset (with analitical check)
32 myhistogram = hist(BW_subset, bins)
33 A = (myhistogram$counts / myhistogram$density)[1] # Normalization factor
34 cairo_pdf('BW_histogram.pdf', width = 16, height = 9, pointsize = 18)
35 plot(myhistogram)
36 curve(A*dcauchy(x,location=x0,scale=HWHM,log=FALSE), lwd=5, add=TRUE, col=mycolor)
37 dev.off()

```

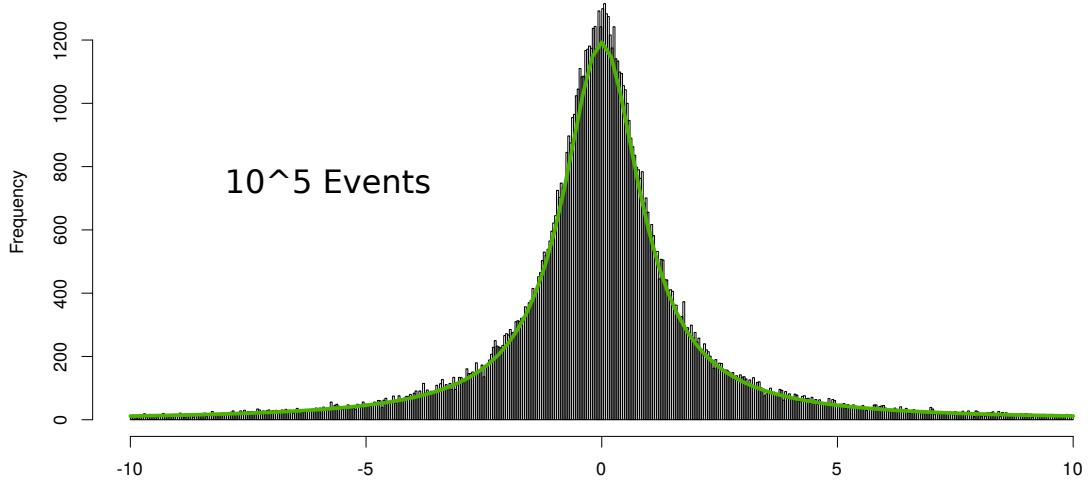
Listing 1: R code for Exercise 1/A (to generate a Breit-Wigner histogram)

Figura 3: Istogrammi generati per $N = 10^5$ eventi, distribuiti in bin di numero crescente.

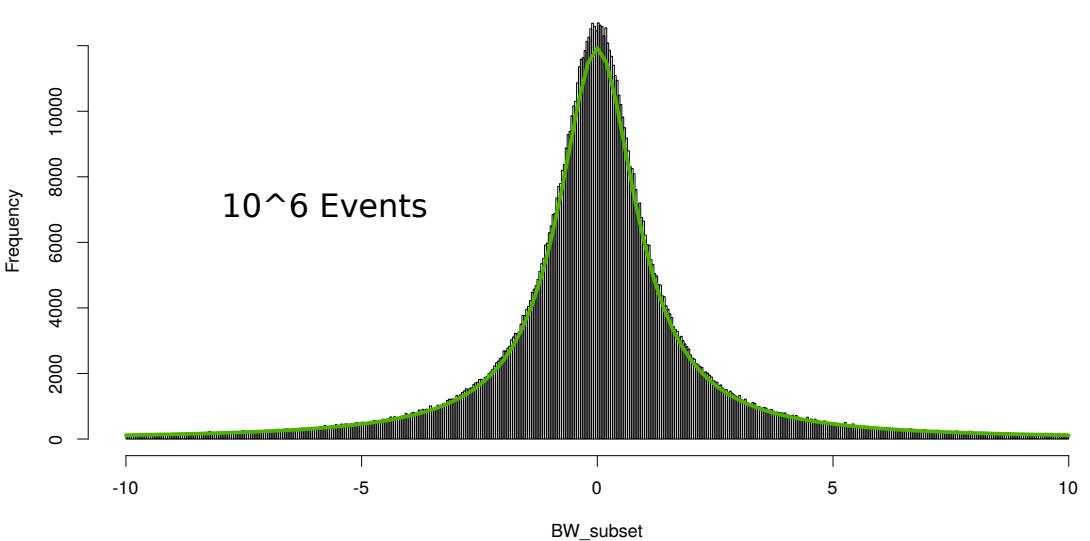


Infine risulta interessante osservare come all'aumentare del numero di bin, fissato $N = 10^5$, aumenti il "rumore" nell'istogramma prodotto (Cfr. Figura 3). Ciò è del tutto ragionevole poiché più sono i bin e più sarà piccolo il campione su cui mediamo il valore da assegnare a ciascun bin. Possiamo quindi concludere che, fissato di converso il numero di bin che ci interessa, riprodurremo meglio la distribuzione tanti più saranno i numeri generati, come ci si aspetta. Una verifica concreta dell'equivalenza di queste due affermazioni si ha osservando la Figura 4, che confronta due istogrammi con medesimo numero di bin (500) ma relativi rispettivamente a 10^5 e 10^6 eventi generati. Tutto ciò costituisce una verifica empirica del Teorema del Limite Centrale.

Figura 4: Istogrammi generati utilizzando 500 bin, ma con differente numero di eventi totali.



(a)



(b)

Passiamo adesso al problema dell'implementazione e del test di un generatore casuale uniforme. L'algoritmo che adotteremo è ad oggi considerato lo standard minimo in termini di prestazioni della sequenza pseudo-random generata: si tratta del *Linear Congruential Method*, nella formulazione di Lewis, Goodman e Miller (Minimal Standard LCG).⁴

In sostanza si tratta di generare la sequenza secondo la seguente formula ricorsiva:

$$\begin{aligned} X_0 &= s \\ X_{t+1} &= (aX_t + b) \bmod c \\ U_t &= \frac{r}{c} \cdot X_t \end{aligned}$$

dove a , b , c e s , sono numeri interi tali che

$$a, b, s \in \{0, 1, \dots, c - 1\},$$

e i numeri $\{U\}$ così costruiti risultano distribuiti uniformemente nell'intervallo aperto⁵ $]0, r[$. Si ricordi che il risultato dell'operazione binaria $q \bmod d$ è il resto della divisione di q per d .

In generale la qualità della sequenza generata dipende dai valori assegnati ai vari parametri (escluso il "seme" s), e i dettagli comportano un notevole livello di complicazione matematica.

La scelta del Minimal Standard LCG

$$\begin{aligned} a &= 7^5 \\ b &= 0 \\ c &= 2^{31} - 1 \end{aligned}$$

pur garantendo buone proprietà di indipendenza alla sequenza $\{U\}$, risulta in un periodo di sole $(2^{31} - 2)$ iterazioni, ormai inadeguato per molte applicazioni d'interesse. L'adozione qui di questo metodo è dunque motivata dalla sola semplicità di implementazione dell'algoritmo.

La possibilità di scegliere semi diversi per diverse simulazioni è molto importante perché permette di ottenere sequenze indipendenti e quindi di combinare statisticamente i risultati ottenuti.⁶ Nel nostro caso tuttavia ci limiteremo a produrre una sola sequenza, impostando $s = 1$: la sequenza risultante è riportata in Figura 5 sottoforma di istogramma delle frequenze dei primi 10^5 numeri generati.

⁴Si vedano ad esempio le *lecture notes*: <https://people.smp.uq.edu.au/DirkKroese/mccourse.pdf>.

⁵Le ragioni per cui tutti i generatori di numeri pseudo-casuali lavorano su intervalli aperti sono di natura computazionale. In sostanza se qualcuno degli U assumesse esattamente uno dei valori estremi si dovrebbero affrontare fastidiosi problemi numerici nell'implementazione di gran parte degli algoritmi che fanno uso di tali generatori.

⁶Nei metodi Montecarlo applicati alla fisica statistica, ad esempio, risulta spesso più utile combinare più simulazioni con sequenze "relativamente brevi" che generare una sola sequenza complessiva. Ciò è dovuto ai processi di minimizzazione coinvolti, che rischiano di rimanere "intrappolati" in minimi locali delle funzioni considerate. Si consulti ad esempio "*Understanding Molecular Simulation*" di Frenkel e Smit (Elsevier 2001).

Minimal Standard LCG Histogram

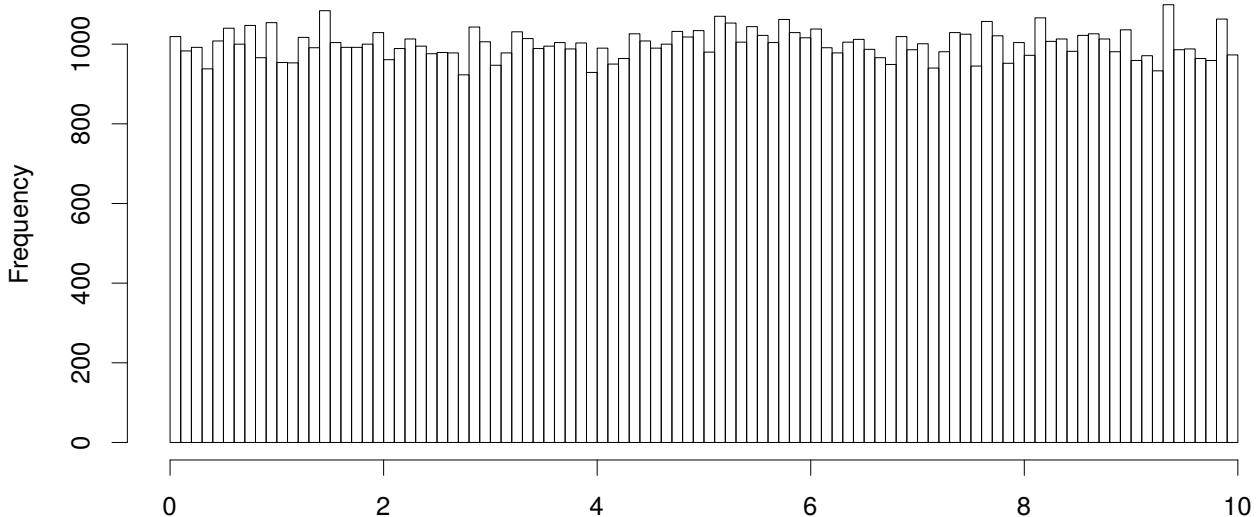


Figura 5: Istogramma di 10^5 numeri generati con algoritmo Minimal Standard LCG, nell'intervallo $]0, 10[$.

```

1 # Defining Minimal Standard parameters for LCG
2 a = 7^5
3 b = 0
4 c = 2^31-1
5
6 # Entering user's choice parameters (length of sequence, interval radius and seed)
7 n = 10^5
8 r = 10
9 s = 1
10
11 # Inizializing recursive vectors
12 x = numeric(length = n)
13 u = numeric(length = n)
14 x[1] = s
15
16 # Recursive cycle
17 t = 1
18 while(t < n)
19 {   u[t] = x[t] * r/c;           #Normalizing the output
20     x[t+1] = (a*x[t] + b) %% c;  #Recursive action (%% is R syntax for mod)
21     t = t+1
22 }; u[t] = x[t] * r/c
23
24 # Saving a histogram plot of {u}
25 cairo_pdf('LCG_histogram.pdf', width = 16, height = 9, pointsize = 22)
26 LCG_hist = hist(u, breaks = n/1000)
27 LCG_hist$name = 'Minimal Standard LCG Histogram'
28 LCG_hist$ylab = 'Frequency'
29 plot(LCG_hist, ylab=LCG_hist$ylab, main=LCG_hist$name)
30 dev.off()

```

Listing 2: R code for Minimal Standard LCG

Il primo test cui possiamo sottoporre i numeri pseudo-casuali appena generati è senz'altro un calcolo della media e della varianza campionarie. Questi sono difatti degli stimatori non distorti per cui ci aspettiamo che, al crescere del numero di estrazioni, convergano ai "valori veri" della distribuzione di probabilità che si vuole simulare.

Cominciamo dunque ricordando le espressioni che definiscono la media e la varianza di una distribuzione uniforme nell'intervallo $[x, y]$:

$$E[u(x, y)] = \frac{x + y}{2} =: \mu(x, y)$$

$$\text{Var}[u(x, y)] = \frac{(y - x)^2}{12} =: \sigma^2(x, y)$$

per cui nel nostro caso $\mu(0, 10) = 5$ e $\sigma(0, 10) = 8.\bar{3}$ saranno i valori di riferimento.

Per quanto riguarda gli stimatori campionari, essi sono definiti come segue:

$$\hat{\mu}(n) := \frac{1}{n} \sum_{t=0}^{n-1} U_t$$

$$\hat{\sigma}^2(n) := \frac{1}{n-1} \sum_{t=0}^{n-1} (U_t - \hat{\mu}(n))^2$$

Dal nostro campione otteniamo risultati soddisfacenti:

$n = 10^5$	valore vero	valore stimato	errore relativo $\times 10^{-3}$
media	5	5.002841...	0.568...
varianza	$8.\bar{3}$	8.319576...	1.651...

da cui deduciamo che il generatore riproduce bene tali proprietà della distribuzione uniforme.

```

1 # Assuming to have a vector u[t], with t = 1,...,n and uniform distributed in [0,r]
2
3 # Sample Mean
4 sm = 0
5 for (t in seq(1, n, by = 1))
6   { sm = sm + u[t] / n }
7
8 # Sample Variance
9 ss = 0
10 for (t in seq(1, n, by = 1))
11   { ss = ss + (u[t] - sm)^2 / (n-1) }
12
13 # Displaying results and relative deviation from 'true values'
14 true_media = r/2;  true_varianza = r^2/12;
15 media = sm;      varianza = ss
16 e_media = abs(media - true_media)/true_media
17 e_varianza = abs(varianza - true_varianza)/true_varianza
18 media; varianza; e_media; e_varianza;
```

Listing 3: R code for computing non-biased variance and mean predictors

Tuttavia il controllo delle sole media e varianza campionarie non può davvero garantire la bontà di una realizzazione campionaria della densità da simulare. Procederemo dunque alla costruzione di opportuni *test d'ipotesi* le cui ipotesi nulle consistano nell'assumere che la sequenza generata segua effettivamente la distribuzione desiderata. Qualora i *p-value* ottenuti consentano di rifiutare tale ipotesi potremmo dire di aver rilevato un problema statisticamente significativo nella sequenza generata.

Test "chi-quadro" alla Pearson

Il primo test considerato è basato sulla cosiddetta statistica "*chi-quadro*", definita come:

$$\chi_{k-1}^2 = \sum_{i=1}^k \frac{(o_i - e_i)^2}{e_i},$$

dove o_i e e_i sono le frequenze rispettivamente osservata e attesa per l'evento E_i , e $k - 1$ sono detti essere i *gradi di libertà* della statistica. Dunque nel nostro caso, diviso l'intervallo $]0, 10[$ in k bin, valuteremo lo scarto quadratico normalizzato tra le frequenze istogrammate dalla sequenza generata e il valore atteso di $(10^5/k)$ eventi su ciascun bin. Evidentemente quanto più la sequenza estratta sarà aderente alla distribuzione uniforme, tanto più il valore ottenuto per la statistica χ^2 sarà tendente a zero. Tra i vari test che possono essere implementati su tale statistica utilizzeremo quello di *Pearson*, nativamente incorporato in R⁷, e ben adatto al caso in analisi⁸.

Lavorando con tre diversi istogrammi dei nostri 10^5 campioni abbiamo ottenuto i seguenti risultati:

Number of bins	Pearson's <i>p-value</i>
500	0.94475363037911164
750	0.86939873721667693
1000	0.85937070391061714

I *p-value* ottenuti sono tali per cui fissato un qualsiasi grado ragionevole di significatività non possiamo rigettare l'ipotesi di una sequenza distribuita uniformemente: il generatore implementato ha superato il test di Pearson.

Test di Kolmogorov-Smirnov

Un altro test d'ipotesi appropriato alla nostra situazione è quello di Kolmogorov e Smirnov (KS), basato sulla funzione cumulante $F(x)$. L'ipotesi nulla sarà dunque formulata in termini di $F(x)$ e della sua candidata realizzazione campionaria $\hat{F}_n(x)$: $H_0 = \{\hat{F}_n(x) = F(x), \forall x\}$. Il test è ben definito solo se $F(x)$ è continua su tutto il dominio. La statistica di riferimento è molto semplice: si tratta dello scarto massimo tra cumulante teorica e sua realizzazione campionaria $D_n = \max |\hat{F}_n(x) - F(x)|$, sul cui valore viene stabilito un *decision boundary* oltre il quale rifiutare H_0 . A tale valore di taglio è associata una significatività α , secondo le formule:

$$D_n^{\text{cut}} = \frac{\lambda_\alpha}{\sqrt{n}},$$

$$1 - \alpha = \sum_{k=-\infty}^{\infty} (-1)^k \exp [-2k^2 \lambda_\alpha].$$

Anche il test KS è nativamente implementato in R⁹ e per i primi 100 numeri della sequenza generata ha restituito: $D_{100} \simeq 0.05954$ e $\lambda_{0.01} \simeq 1.6276$. Dunque avendo $D_{100} < D_{100}^{\text{cut}} = 0.16276$ possiamo affermare che, fissata una significatività del 1%, anche il test KS conferma la bontà dal nostro generatore LCG.

⁷Comando `chisq.test`: <https://stat.ethz.ch/R-manual/R-devel/library/stats/html/chisq.test.html>

⁸Il test di Pearson presenta problemi qualora alcune delle frequenze attese siano molto piccole, per cui per distribuzioni con "lunghe code" potrebbe risultare inutilizzabile. Di certo una densità uniforme non presenta criticità in tal senso.

⁹Comando `ks.test`: <https://stat.ethz.ch/R-manual/R-devel/library/stats/html/ks.test.html>

Infine osserviamo che sebbene il superamento dei test d'ipotesi proposti dia una conferma molto solida su quanto la sequenza generata riproduca la distribuzione da simulare, resta ancora la possibilità che una sequenza palesemente deterministica, e quindi affetta da spiacevoli correlazioni, minimizzi molto efficientemente gli scarti sulla $f(x)$ e sulla $F(x)$ che i test vanno a controllare. Per scongiurare questa possibilità si potrebbero ripetere i test molte volte, variando in un grande range la dimensione dei bin utilizzati, ma questo evidentemente implica un grande dispendio di risorse computazionali. Quindi preferiamo affidarci ad un metodo di controllo visivo e immediato per scovare eventuali correlazioni fra coppie di numeri estratti consecutivamente¹⁰: costruiamo uno *scatter-plot* bidimensionale in cui in ascissa sono riportati i numeri estratti ai passi dispari e in ordinata i numeri estratti ai passi pari. Il risultato, visibile in Figura 6, non mostra nessun evidente pattern distinguibile da un generico e omogeneo "rumore bianco", per cui possiamo concludere che il generatore non presenta problemi di eccessiva correlazione nella sequenza prodotta.

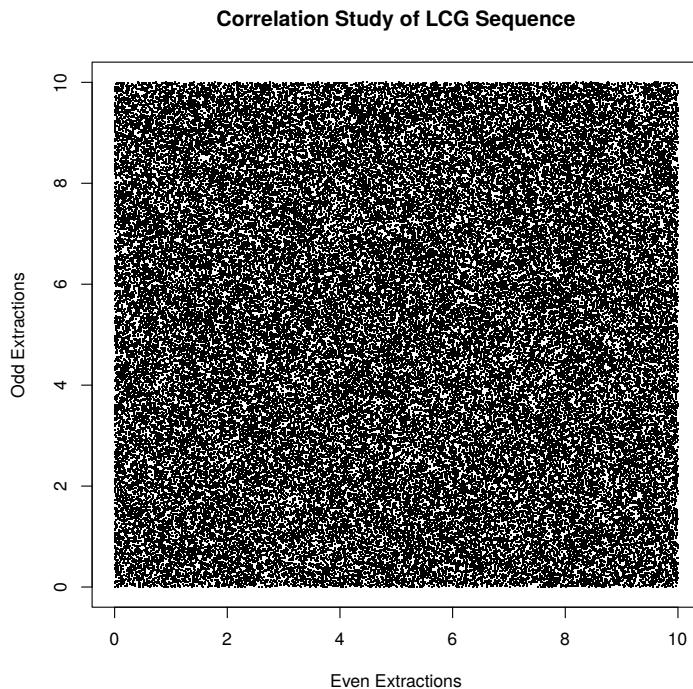


Figura 6: Visualizzazione delle eventuali correlazioni fra numeri estratti consecutivamente dal generatore LCG.

```

1 # Assuming to have a vector u[t], of uniform distributed pseudo-random numbers...
2
3 # Subset of "even extractions"
4 X = u[c(TRUE, FALSE)]
5 # Subset of "odd extractions"
6 Y = u[c(FALSE, TRUE)]
7 # Saving a scatter-plot of Y vs X
8 cairo_pdf('CorrelationStudy.pdf')
9 Xlab = 'Even Extractions'
10 Ylab = 'Odd Extractions'
11 Title = 'Correlation Study of LCG Sequence'
12 plot(X, Y, pch = '.', xlab = Xlab, ylab = Ylab, main = Title)
13 dev.off()

```

Listing 4: R code for correlation-study of LCG random sequence

¹⁰Va da sé che andrebbero controllate anche le correlazioni fra numeri estratti non consecutivamente, ma ci aspettiamo che gli effetti di correlazione diminuiscano fortemente al crescere del numero di "passi" che ne separano l'estrazione nell'algoritmo del generatore (purché si stia ben sotto il periodo di ricorrenza!).

Esercizio 2

Si implementi un programma per il calcolo della costante di Eulero-Mascheroni, definita per mezzo dell'Eq.(1), discutendo l'influenza degli errori di calcolo sul risultato ottenuto.

$$\gamma = \lim_{n \rightarrow \infty} \left(\sum_{k=1}^n \frac{1}{k} - \log n \right) \quad (1)$$

* * *

Ad un primo sguardo il problema principale del calcolo appare insito nella somma del numero armonico:

$$H_n = \sum_{k=1}^n \frac{1}{k}.$$

Difatti, come per qualsiasi somma in cui i termini decrescano progressivamente, è necessario implementare alcune accortezze per scongiurare il rischio di errori di troncamento (*round-off*). In particolare comandando, come da implementazione letterale della definizione, i termini in ordine decrescente si rischia dopo un alto numero di iterazioni di trovarsi a sommare numeri di ordini di grandezza decisamente diversi: da un lato una somma parziale ormai "grande" e dall'altro un k -esimo termine via via sempre più "piccolo". Pur utilizzando variabili *floating point* con precisione doppia¹¹ ciò comporta, per n sufficientemente grande, il troncamento di cifre significative dell'addendo minore e, sebbene su una singola operazione tale arrotondamento sia in genere quantitativamente accettabile, nell'iterazione di più somme può portare a risultati completamente errati.¹²

La situazione può essere migliorata con la semplice inversione dell'ordine di somma: iniziando dai termini di ordine minimo la somma parziale crescerà molto lentamente, il tutto a fronte di addendi man mano sempre più grandi. Risulta dunque decisamente più improbabile trascurare termini a causa dei limiti di rappresentazione del calcolatore.

La rilevanza dell'ordine di somma è stata verificata calcolando la somma per $n = 50$ con entrambi gli algoritmi e confrontando i risultati con il valore che Wolfram-Alpha¹³ riporta per H_{50} . Di seguito si riportano il codice utilizzato e i risultati relativi.

```
1 # Entering the desired order for the harmonic sum
2 n = 50
3
4 # Direct: literal transposition of sum in for cycle
5 H = 0
6 for (k in seq(1,n, by = 1))
7 { H = H + 1/k }
8 print(H, digits = 16)
9
10 # Trick: invert sum order (avoids round-offs!)
11 H = 0
12 k = n
13 while (k > 0)
14 { H = H + 1/k; k = k - 1 }
15 print(H, digits = 16)
```

Listing 5: R code for summing harmonic numbers

¹¹Opzione di default del linguaggio R.

¹²In ultima analisi tutto dipende da quanto velocemente "decade" la coda della somma.

¹³<https://www.wolframalpha.com>

Wolfram-Alpha [16 digits]	$H_{50} \simeq 4.49920533832942\check{5}$
Direct algorithm [16 digits]	$H_{50} \simeq 4.49920533832942\check{3}$
Inverted sum [16 digits]	$H_{50} \simeq 4.49920533832942\check{5}$

Il vantaggio ottenuto con l'inversione dell'ordine degli addendi permette di guadagnare una cifra significativa rispetto all'implementazione diretta della somma, come risulta evidente osservando in tabella le cifre evidenziate.

Tuttavia implementando di conseguenza il calcolo di γ osserviamo degli errori che limitano ancora severamente la precisione ottenibile: nonostante la somma invertita dia un risultato più piccolo dell'algoritmo diretto (e quindi più vicino al valore vero, che li minora entrambi) non riusciamo in nessuno dei due casi a fissare più di quattro cifre significative.

"True value" [First 16 digits from O.E.I.S.]	$\gamma \simeq 0.5772\check{1}56649015329$	$ \gamma_{\text{true}} - \gamma_{\text{calc.}} $ [1 digit]
Direct algorithm [$n = 10^5$ 16 digits]	$\gamma \simeq 0.5772\check{2}06648932138$	5e-06
Inverted sum [$n = 10^5$ 16 digits]	$\gamma \simeq 0.5772\check{2}06648931792$	5e-06

```

1 # Setting Eulero-Mascheroni constant (correct up to 16 digits)
2 EM = 0.5772156649015329
3
4 # Entering the desired truncation-parameter for the series
5 n = 10^5
6
7 # First attempt: direct algorithm from definition
8 em = -log(n)
9 for (k in seq(1,n, by = 1))
10 { em = em + 1/k }
11 print(EM, digits = 16)
12 print(em, digits = 16)
13 print(abs(EM-em), digits = 1)
14
15 # Second attempt: inverted sum order
16 em = 0
17 k = n
18 while (k > 0)
19 { em = em + 1/k; k = k - 1 }
20 em = em-log(n)
21 print(EM, digits = 16)
22 print(em, digits = 16)
23 print(abs(EM-em), digits = 1)
24 [...]

```

Listing 6: R code for naïve computation of Eulero-Mascheroni constant

Tornando all'equazione (1) notiamo infatti che pur costruendo in maniera ottimizzata il numero armonico H_n , sottraendogli "tutto insieme" il logaritmo di n ci si ritrova ancora in una situazione spiacevole: si stanno sostanzialmente sottraendo tra loro due numeri "grandi" - dello stesso ordine in n - allo scopo di ottenere un risultato dell'ordine dell'unità e quindi certamente svariati ordini di grandezza sotto n ; ne consegue che per le cifre significative del risultato utilizziamo solo una piccola parte dei *bit* di rappresentazione disponibili.

Per risolvere il problema l'idea è quindi di scomporre in addendi anche il logaritmo di n in modo da costruire γ sommando (algebricamente) termini il più possibile confrontabili con la somma parziale. Sebbene la successione

logaritmica non abbia uno sviluppo in serie immediato, è possibile raggiungere l'obiettivo auspicato con delle opportune manipolazioni:

Definendo la successione

$$\gamma_n = H_n - \log n,$$

possiamo scrivere

$$\begin{aligned}\gamma_n &= H_n - k \frac{\log n}{k} \quad \forall k > 0 \\ &= H_n - \frac{\log n^k}{k} \\ &= H_n - \frac{H_{n^k} - \gamma_{n^k}}{k} \iff \gamma_n - \frac{\gamma_{n^k}}{k} = H_n - \frac{H_{n^k}}{k},\end{aligned}$$

per cui, dal momento che $\lim_{n \rightarrow \infty} \gamma_n \equiv \lim_{n \rightarrow \infty} \gamma_{n^k} = \gamma$, si ha $\gamma = \frac{k}{k-1} \lim_{n \rightarrow \infty} \left(H_n - \frac{H_{n^k}}{k} \right)$.

Scegliendo in particolare $k = 2$ otteniamo una semplice formula, dovuta al matematico russo Mačys¹⁴:

$$\begin{aligned}\gamma &= \lim_{n \rightarrow \infty} (2H_n - H_{n^2}) \\ &= \lim_{n \rightarrow \infty} \left(1 + \frac{1}{2} + \cdots + \frac{1}{n} - \frac{1}{n+1} - \cdots - \frac{1}{n^2} \right).\end{aligned}\tag{2}$$

La formula di Mačys ben si presta ad un'implementazione analoga a quella utilizzata per il calcolo ottimizzato del numero armonico H_n e ci permette di abbassare di più di cinque ordini di grandezza l'errore $|\gamma_{\text{true}} - \gamma_{\text{calc.}}|$ - da circa $5 \cdot 10^{-6}$ a circa 10^{-11} - portando fino a nove le cifre significative corrette:

$$\begin{aligned}\gamma_{\text{true}} &= 0.577215664\check{9}015329\dots \\ \gamma_{\text{Macys}} &\simeq 0.577215664\check{8}885598\dots\end{aligned}$$

Infine la formula di Mačys può essere resa più accurata valutando l'errore commesso troncando a n finiti per mezzo di sviluppi asintotici del numero armonico H_n . Senza riportarne esplicitamente la derivazione utilizziamo la correzione¹⁵

$$\begin{aligned}\gamma &= \lim_{n \rightarrow \infty} \left(1 + \frac{1}{2} + \cdots + \frac{1}{n} - \frac{1}{n+1} - \cdots - \frac{1}{n^2} - \underbrace{\frac{1}{n^2+1} - \cdots - \frac{1}{n^2+n}}_{\text{1st Trick}} + \overbrace{\frac{1}{6n^2} - \frac{1}{6n^3}}^{\text{2nd Trick}} \right) \\ &= \lim_{n \rightarrow \infty} \left(2H_n - H_{n(n+1)} + \frac{1}{6n^2} - \frac{1}{6n^3} \right),\end{aligned}\tag{3}$$

il cui errore per n finito può essere dimostrato essere dell'ordine $O(n^{-4})$.

¹⁴J. J. Mačys, *On the Euler–Mascheroni Constant*, Mat. Zametki, 94:5 (2013), 695–701 or Math. Notes, 94:5 (2013), 647–652
¹⁵Maggiori dettagli possono essere trovati consultando la seguente discussione ([link](#)) sul forum [math.stackexchange.com](#)

Ne risulta il guadagno di altre due cifre significative corrette, con un errore assoluto di circa $4 \cdot 10^{-12}$. Notiamo subito che tale valore è molto maggiore di $n^{-4} = 10^{-20}$, segno che la precisione è ancora limitata da errori di troncamento sulle singole somme. Tuttavia il lavoro di ottimizzazione dell'algoritmo ha permesso complessivamente di abbattere di ben sei ordini di grandezza l'errore sul calcolo, rispetto all'implementazione pedissequa della formula (1). In ragione di ciò riteniamo il risultato ottenuto pienamente soddisfacente.

In conclusione riportiamo una tabella comparativa dei risultati relativi ai quattro algoritmi utilizzati e a seguire il codice per l'implementazione delle formule di Mačys.

"True value" [First 16 digits from O.E.I.S.]	$\gamma \simeq 0.5772156649015329$	$ \gamma_{\text{true}} - \gamma_{\text{calc.}} $ [1 digit]
Direct algorithm [$n = 10^5$ 16 digits]	$\gamma \simeq 0.5772\check{2}06648932138$	5e-06
Inverted sum [$n = 10^5$ 16 digits]	$\gamma \simeq 0.5772\check{2}06648931792$	5e-06
Mačys Formula [$n = 10^5$ 16 digits]	$\gamma \simeq 0.577215664\check{8}885598$	1e-11
Mačys + Tricks [$n = 10^5$ 16 digits]	$\gamma \simeq 0.57721566490\check{5}2262$	4e-12

```

1 ...
2 # Third attempt: Macys formula
3 em = 0
4 k = n^2
5 while (k > n)
6 { em = em - 1/k; k = k - 1 }
7 while (k > 0)
8 { em = em + 1/k; k = k -1 }
9 print(EM, digits = 16)
10 print(em, digits = 16)
11 print(abs(EM-em), digits = 1)
12
13 # Final refinement: some magic
14 em = 0
15 k = n^2+n # 1st Trick
16 while (k > n)
17 { em = em - 1/k; k = k - 1 }
18 while (k > 0)
19 { em = em + 1/k; k = k - 1 }
20 em = em + 1/(6*n^2) - 1/(6*n^3) # 2nd Trick
21 print(EM, digits = 16)
22 print(em, digits = 16)
23 print(abs(EM-em), digits = 1)

```

Listing 7: R code for efficient computation of Eulero-Mascheroni constant

Esercizio 3

Si scriva in un linguaggio di programmazione opportuno una procedura per la minimizzazione col metodo del simplesso e la si utilizzi per determinare il minimo globale o i minimi locali di:

- Funzione di Easom

$$f(x, y) = -\cos(x) \cos(y) \exp[-(x - \pi)^2 - (y - \pi)^2]$$

- Funzione di Goldstein-Price

$$f(x, y) = \exp[0.5(x^2 + y^2 - 25)^2] + \sin^4(4x - 3y) + 0.5(2x + y - 10)^2$$

- Funzione di Bukin Nr. 6

$$f(x, y) = 0.01|x + 10| + 100\sqrt{|y - 0.01x^2|}$$

- Funzione di Booth

$$f(x, y) = (x + 2y - 7)^2 + (2x + y - 5)^2$$

discutendo i risultati ottenuti.

* * *

Il metodo Nelder-Mead (anche detto del *del simplesso* o *metodo ameba*) è un algoritmo non lineare per la minimizzazione di funzioni di n variabili senza l'uso delle derivate. La nozione centrale è quella di simplesso n -dimensionale, definito come un politopo convesso con $n + 1$ vertici¹⁶.

Consideriamo una generica funzione $f : \mathbb{R}^n \mapsto \mathbb{R}$, sufficientemente regolare, della quale ci interessano i minimi locali e globali. L'idea di Nelder e Mead consiste nel definire un generico simplesso di prova¹⁷ e valutare tale funzione sui vertici di tale politopo. Definiti questi come $\mathbf{x}_i \in \mathbb{R}^n$, $i = 1 \dots n + 1$, si otterrà allora la sequenza di valori $\{f(\mathbf{x}_i)\}$; i vertici sono da intendersi ordinati per soddisfare $f(\mathbf{x}_1) \leq \dots \leq f(\mathbf{x}_{n+1})$. A questo punto si ipotizza euristicamente che il vertice \mathbf{x}_{n+1} sia quello con minore probabilità di trovarsi in prossimità di un qualche minimo di $f(\mathbf{x})$ e si procede a valutarne un sostituto, con la speranza che quest'ultimo produca un valore inferiore rispetto al *penultimo* vertice, una volta valutata in esso la funzione. Come primo tentativo si valuta dunque una *riflessione* dell'ultimo vertice della sequenza rispetto al *centroide* $\bar{\mathbf{x}}$ degli altri n vertici, ossia:

$$\mathbf{x}_r = \bar{\mathbf{x}} + \mu_r(\bar{\mathbf{x}} - \mathbf{x}_{n+1}), \quad \bar{\mathbf{x}} = \frac{1}{n} \sum_{k=1}^n \mathbf{x}_k, \quad \mu_r > 0,$$

Qualora tale mossa non avesse successo, ossia se $f(\mathbf{x}_r) \geq f(\mathbf{x}_n)$, si prova a sostituire uno degli altri punti, con una qualche riduzione del volume del simplesso. L'idea è in definitiva che il simplesso evolva per riflessioni, espansioni e contrazioni¹⁸ fino ad individuare una direzione di gradiente e così raggiungere velocemente un minimo locale della funzione f .

Esistono vari modi per implementare un metodo del simplesso che sia concretamente robusto (e magari efficiente). L'algoritmo che noi presentiamo prevede le seguenti mosse gerarchiche:

1. *Ordinamento* dei vertici secondo il valore che in essi assume la funzione obiettivo:

$$f(\mathbf{x}_1) \leq \dots \leq f(\mathbf{x}_i) \leq \dots \leq f(\mathbf{x}_{n+1}).$$

2. Valutazione della *tolleranza* raggiunta sui valori $f(\mathbf{x}_i)$: se $|f(\mathbf{x}_{n+1}) - f(\mathbf{x}_1)| \leq \varepsilon \ll 1$ o è stato individuato un minimo o il simplesso è entrato in una regione di *plateau*; in entrambi i casi si chiude la routine. Se invece si è al di sopra della tolleranza stabilita ε si prosegue con il punto successivo.

¹⁶Per politopo si intende la generalizzazione a n qualsiasi del poligono ($n = 2$) e del poliedro ($n = 3$). La nozione di convessità è l'immediata estensione di quella in bassa dimensione.

¹⁷In realtà la scelta del primo insieme di vertici è decisiva sull'efficienza pratica del metodo, ragion per cui sarà opportuno valutarla attentamente.

¹⁸Da qui il riferimento alle amebe!

3. Calcolo del *centroide* $\bar{\mathbf{x}}$ dei primi n vertici e del candidato sostituto per \mathbf{x}_{n+1} : $\mathbf{x}_r = \mathbf{x}_r(\mu_r)$.
4. *Riflessione* effettiva di \mathbf{x}_{n+1} in \mathbf{x}_r se quest'ultimo risulta migliore di \mathbf{x}_n ma non più valido di \mathbf{x}_1 , ossia se $f(\mathbf{x}_1) \leq f(\mathbf{x}_r) < f(\mathbf{x}_n)$. In tale caso si ritorna al punto 1. Altrimenti si procede con i punti seguenti.
5. *Espansione* del simplesso. Se il punto riflesso \mathbf{x}_r risulta migliore di tutti i vertici, secondo $f(\mathbf{x}_r) < f(\mathbf{x}_1)$, si procede a calcolare il *punto espanso* $\mathbf{x}_e = \bar{\mathbf{x}} + \mu_e(\bar{\mathbf{x}} - \mathbf{x}_{n+1})$, $\mu_e > \mu_r$ e a valutare in esso la funzione f . Se $f(\mathbf{x}_e) < f(\mathbf{x}_r)$ vale la pena di allungare il simplesso nella direzione di riflessione fino a \mathbf{x}_e , che dunque viene sostituito a \mathbf{x}_{n+1} . Se invece $f(\mathbf{x}_e) > f(\mathbf{x}_r)$ ci si accontenta di sostituire \mathbf{x}_r a \mathbf{x}_{n+1} . In entrambi i casi si ritorna al punto 1.
6. *Contrazione*. Resta ancora il caso in cui $f(\mathbf{x}_r) \geq f(\mathbf{x}_n)$. In tale situazione risulta necessario contrarre la dimensione del simplesso, perché presumibilmente confinato dentro una "valle" angusta rispetto alle sue dimensioni. Distinguiamo due casi: se addirittura $f(\mathbf{x}_r) \geq f(\mathbf{x}_{n+1})$ sarà necessario operare una "contrazione interna":

$$\mathbf{x}_c^{\text{int}} = \bar{\mathbf{x}} + \mu_c^{\text{int}}(\bar{\mathbf{x}} - \mathbf{x}_{n+1}), \quad -1 \leq \mu_c^{\text{int}} < 0;$$

se invece $f(\mathbf{x}_n) \leq f(\mathbf{x}_r) < f(\mathbf{x}_{n+1})$ si può contrarre andando comunque a riflettere oltre il centroide, in una cosiddetta "contrazione esterna":

$$\mathbf{x}_c^{\text{ext}} = \bar{\mathbf{x}} + \mu_c^{\text{ext}}(\bar{\mathbf{x}} - \mathbf{x}_{n+1}), \quad 0 < \mu_c^{\text{ext}} < \mu_r,$$

ottenendo un vantaggio sulla velocità complessiva di convergenza della routine. In entrambi i casi si valuta la diseguaglianza $f(\mathbf{x}_c) < f(\mathbf{x}_r)$: se soddisfatta si sostituisce il punto contratto a \mathbf{x}_{n+1} e si torna al punto 1; altrimenti non resta altro che abbandonare la direzione individuata da \mathbf{x}_{n+1} e $\bar{\mathbf{x}}$ e cercarne un'altra operando sugli altri vertici secondo il punto seguente.

7. *Riduzione*. Nel raro caso in cui la contrazione aumenti il valore di f si procede a contrarre tutti i vertici tranne \mathbf{x}_1 , secondo la:

$$\mathbf{x}_i = \mathbf{x}_1 + \mu_c^{\text{red}}(\mathbf{x}_i - \mathbf{x}_1), \quad \forall i \in \{2 \dots n\}.$$

Segue invariabilmente il passo 1.

Da quel che abbiamo visto risulta che la routine prevede 5 parametri liberi, da scegliere in base alle esigenze del singolo caso. I valori più comunemente (e da noi) impiegati sono:

$$\mu_r = 1 \quad \mu_e = 2$$

$$\mu_c^{\text{ext}} = 0.5 \quad \mu_c^{\text{int}} = -0.5$$

$$\mu_c^{\text{red}} = 0.5$$

Di seguito riportiamo il codice che implementa in Phyton l'algoritmo appena illustrato:

```

1 from pylab import *
2 import numpy as np
3 import operator
4
5 ## Search-Routine Parameters
6 n = 2
7 mu_exp = 2.0
8 mu_rifl = 1.0
9 mu_contr_ex = 1.0/2
10 mu_contr_int = -1.0/2
11 mu_red = 1.0/2
12
13 ## Random Trial Simplex [may need a careful range-definition]
14 def vertici_iniziali():
15     import random
16     random.seed()
17     vertici=[]
18     for i in range(n+1):
19         x=random.uniform(-12,-8);
20         y=random.uniform(0,3);
21         vertici.append([x,y])
22     return vertici
23 vertex=vertici_iniziali()
24 x1=vertex[0]
25 x2=vertex[1]
26 x3=vertex[2]
27
28 ## Target Functions [uncomment the desired one]
29 def f(x,y):
30     # EASOM
31     #return -1.0*np.cos(x)*np.cos(y)*np.exp(-((x-np.pi)**2+(y-np.pi)**2))
32     # GOLDSTEIN-PRICE
33     #return np.exp(0.5*(x**2+y**2-25)**2)+(np.sin(4*x-3*y))**4+0.5*(2*x+y-10)**2
34     # BUKIN 6th
35     #return 100*abs(y-0.01*x**2)+0.01*abs(x+10)
36     # BOOTH
37     #return (x+2*y-7)**2+(2*x+y-5)**2
38
39 data_x=[x1,x2,x3]
40 data_f=[f(x1[0],x1[1]),f(x2[0],x2[1]),f(x3[0],x3[1])]
41 data=[[f(x1[0],x1[1]),x1],[f(x2[0],x2[1]),x2],[f(x3[0],x3[1]),x3]]
42 print (data)
43
44 ## Ordering [increasing f(x)]
45 data=sorted(data,key=operator.itemgetter(0))
46 data_f= [item[0] for item in data]
47 data_x= [item[1] for item in data]
48 print (data_f, 'f(trial simplex)')
49 print ( data_x , 'trial simplex')
50
51 ## Plotting the Target Function and the Trial Simplex
52 xvec = np.linspace(-12, -8, 1000)
53 yvec = np.linspace(-0, 3, 1000)
54 X,Y = np.meshgrid(xvec, yvec)
55 Z = f(X, Y).T
56 fig, ax = subplots()
57 im = imshow(Z, cmap=cm.magma, vmin=Z.min(), vmax=Z.max(), extent=[-12, -8, 0, 3])

```

```

58 im.set_interpolation('bilinear')
59 cb = fig.colorbar(im)
60 Xvertex = np.array([])
61 Yvertex = np.array([])
62 for i in range(n+1):
63     Xvertex = np.append(Xvertex, data_x[i][0])
64     Yvertex = np.append(Yvertex, data_x[i][1])
65 plt.scatter(Xvertex,Yvertex, color='green', edgecolor='black', s=200)
66 coord = data_x
67 coord.append(coord[0]) # have to repeat the first point to create a 'closed loop'
68 xs, ys = zip(*coord) # creates lists of x and y values
69 plt.plot(xs,ys, color='white', alpha=0.3, ls='--') # Polytope draws up
70
71 ## Evolving the Simplex
72 epsilon = 10**(-5) # See...
73 loop=1
74 while data_f[n]- data_f[0] > epsilon: # ...this!
75
76     loop=loop+1
77
78     ## Centroid
79     a=np.array(data_x[0:n])
80     a=a/n
81     xc=a.sum(axis=0)
82     xr=(1+mu_rifl)*xc-mu_rifl*np.array(data_x[n])
83     fr=f(xr[0],xr[1])
84     print (' fr, ',fr)
85
86     ## Reflection step
87     if data_f[0]<=fr<data_f[n-1]:
88         data[n][1]=xr
89         data[n][0]=f(xr[0],xr[1])
90         data=sorted(data,key=operator.itemgetter(0))
91         print ('loop',loop,'riflessione')
92         data_f= [item[0] for item in data]
93         data_x= [item[1] for item in data]
94         print (data_f, 'valori funzione ')
95         print ( data_x , 'vertici ')
96         for i in range(n+1):
97             Xvertex = np.append(Xvertex, data_x[i][0])
98             Yvertex = np.append(Yvertex, data_x[i][1])
99             plt.scatter(Xvertex,Yvertex, color='white', edgecolor='black')
100            coord = data_x
101            coord.append(coord[0]) # repeat the first point to create a 'closed loop'
102            xs, ys = zip(*coord) # create lists of x and y values
103            plt.plot(xs,ys, color='white', alpha=0.3, ls='--')
104            continue
105
106     ## Expansion step
107     if fr<data_f[0]:
108         a=np.array(data_x[0:n])
109         a=a/n
110         xc=a.sum(axis=0)
111         xe=(1+mu_exp)*xc-mu_exp*np.array(data_x[n])
112         fe=f(xe[0],xe[1])
113         if fe<fr:
114             data[n][1]=xe
115             data[n][0]=f(xe[0],xe[1])
116             data=sorted(data,key=operator.itemgetter(0))

```

```

117     print('loop' ,loop , 'espansione')
118     data_f= [item[0] for item in data]
119     data_x= [item[1] for item in data]
120     print (data_f , 'valori funzione ')
121     print ( data_x , 'vertici ')
122     for i in range(n+1):
123         Xvertex = np.append(Xvertex, data_x[i][0])
124         Yvertex = np.append(Yvertex, data_x[i][1])
125     plt.scatter(Xvertex,Yvertex, color='white', edgecolor='black')
126     coord = data_x
127     coord.append(coord[0]) # repeat the first point to create a 'closed loop'
128     xs, ys = zip(*coord) # create lists of x and y values
129     plt.plot(xs,ys, color='white', alpha=0.3, ls='--')
130     continue
131 else:
132     data[n][1]=xr
133     data[n][0]=f(xr[0],xr[1])
134     data=sorted(data,key=operator.itemgetter(0))
135     print('loop',loop,'riflessione')
136     data_f= [item[0] for item in data]
137     data_x= [item[1] for item in data]
138     print (data_f , 'valori funzione ')
139     print ( data_x , 'vertici ')
140     for i in range(n+1):
141         Xvertex = np.append(Xvertex, data_x[i][0])
142         Yvertex = np.append(Yvertex, data_x[i][1])
143     plt.scatter(Xvertex,Yvertex, color='white', edgecolor='black')
144     coord = data_x
145     coord.append(coord[0]) # repeat the first point to create a 'closed loop'
146     xs, ys = zip(*coord) # create lists of x and y values
147     plt.plot(xs,ys, color='white', alpha=0.3, ls='--')
148     continue
149
150 ## External -Contraction step
151 if data_f[n-1]<=fr<data_f[n]:
152     a=np.array(data_x[0:n])
153     a=a/n
154     xc=a.sum(axis=0)
155     xoc=(1+mu_contr_ex)*xc-mu_contr_ex*np.array(data_x[n])
156     foc=f(xoc[0],xoc[1])
157
158 if foc<fr:
159     data[n][1]=xoc
160     data[n][0]=f(xoc[0],xoc[1])
161     data=sorted(data,key=operator.itemgetter(0))
162     print( 'loop' ,loop , 'contrazione esterna')
163     data_f= [item[0] for item in data]
164     data_x= [item[1] for item in data]
165     print (data_f , 'valori funzione ')
166     print ( data_x , 'vertici ')
167     for i in range(n+1):
168         Xvertex = np.append(Xvertex, data_x[i][0])
169         Yvertex = np.append(Yvertex, data_x[i][1])
170     plt.scatter(Xvertex,Yvertex, color='white', edgecolor='black')
171     coord = data_x
172     coord.append(coord[0]) # repeat the first point to create a 'closed loop'
173     xs, ys = zip(*coord) # create lists of x and y values
174     plt.plot(xs,ys, color='white', alpha=0.3, ls='--')
175     continue

```

```

176
177     ## Reduction step []
178     else:
179         a=np.array(data_x)
180         for i in range(1,n+1):
181             data[i][1]=a[0]+mu_red*(a[i]-a[0])
182             data[i][0]=f(data[i][1][0],data[i][1][1])
183             data=sorted(data,key=operator.itemgetter(0))
184             print('loop',loop,'riduzione')
185             data_f= [item[0] for item in data]
186             data_x= [item[1] for item in data]
187             print (data_f, 'valori funzione ')
188             print ( data_x , 'vertici ')
189             for i in range(n+1):
190                 Xvertex = np.append(Xvertex, data_x[i][0])
191                 Yvertex = np.append(Yvertex, data_x[i][1])
192             plt.scatter(Xvertex,Yvertex, color='white', edgecolor='black')
193             coord = data_x
194             coord.append(coord[0]) # repeat the first point to create a 'closed loop'
195             xs, ys = zip(*coord) # create lists of x and y values
196             plt.plot(xs,ys, color='white', alpha=0.3, ls='--')
197             continue
198
199     ## Internal-Contraction step
200     if fr>=data_f[n]:
201         a=np.array(data_x[0:n])
202         a=a/n
203         xc=a.sum(axis=0)
204         xic=(1+mu_contr_int)*xc-mu_contr_int*np.array(data_x[n])
205         fic=f(xic[0],xic[1])
206         if fic<data_f[n]:
207             data[n][1]=xic
208             data[n][0]=f(xic[0],xic[1])
209             data=sorted(data,key=operator.itemgetter(0))
210             print('loop',loop , 'contrazione interna')
211             data_f= [item[0] for item in data]
212             data_x= [item[1] for item in data]
213             print (data_f, 'valori funzione ')
214             print ( data_x , 'vertici ')
215             for i in range(n+1):
216                 Xvertex = np.append(Xvertex, data_x[i][0])
217                 Yvertex = np.append(Yvertex, data_x[i][1])
218             plt.scatter(Xvertex,Yvertex, color='white', edgecolor='black')
219             coord = data_x
220             coord.append(coord[0]) # repeat the first point to create a 'closed loop'
221             xs, ys = zip(*coord) # create lists of x and y values
222             plt.plot(xs,ys, color='white', alpha=0.3, ls='--')
223             continue
224
225     ## Reduction step []
226     else:
227         a=np.array(data_x)
228         for i in range(1,n+1):
229             data[i][1]=a[0]+mu_red*(a[i]-a[0])
230             data[i][0]=f(data[i][1][0],data[i][1][1])
231             data=sorted(data,key=operator.itemgetter(0))
232             print('loop',loop , 'riduzione')
233             data_f= [item[0] for item in data]
234             data_x= [item[1] for item in data]

```

```

235     print (data_f, 'valori funzione ')
236     print ( data_x ,'vertici ')
237     for i in range(n+1):
238         Xvertex = np.append(Xvertex, data_x[i][0])
239         Yvertex = np.append(Yvertex, data_x[i][1])
240         plt.scatter(Xvertex,Yvertex, color='white', edgecolor='black')
241         coord = data_x
242         coord.append(coord[0]) # repeat the first point to create a 'closed loop'
243         xs, ys = zip(*coord) # create lists of x and y values
244         plt.plot(xs,ys, color='white', alpha=0.3, ls='--')
245         continue
246
247 ## Plotting the Final Simplex [a single point if the routine has converged!]
248 Xvertex = np.array([])
249 Yvertex = np.array([])
250 for i in range(n+1):
251     Xvertex = np.append(Xvertex, data_x[i][0])
252     Yvertex = np.append(Yvertex, data_x[i][1])
253 plt.scatter(Xvertex,Yvertex, color='red', edgecolor='black', s=100)
254
255 ## Showing all the plots...
256 plt.show()

```

Listing 8: Python code for Simplex Minimization Routine

La nostra implementazione del metodo Nelder-Mead è stata applicata alle quattro funzioni proposte, con estrazione casuale delle coordinate del simplex iniziale all'interno di regioni scelte di volta in volta: per la generica ricerca di minimi locali sono stati utilizzati intervalli piuttosto ampi in x e y , comparabili ai *domini di ricerca* riportati in letteratura¹⁹, mentre per l'ottimizzazione globale si è cercato di restringere opportunamente il campo, attorno al minimo di interesse. Di seguito riportiamo i risultati ottenuti in forma per lo più grafica: *colormap-plot* delle funzioni obiettivo con sovrapposti gli step di evoluzione del simplex; in verde sono evidenziati i vertici iniziali estratti e in rosso il punto di convergenza finale della routine.

Funzione di Easom

Osservando il grafico della funzione di Easom riportato in Figura 7 deduciamo che presenta un solo minimo globale, ben individuabile all'interno di un sostanziale *plateau*. Le sue coordinate sono (π, π) e il valore assunto è -1 .

L'algoritmo riesce a individuarne la posizione con facilità, anche a partire da simplex non posizionati nella sua immediata prossimità: le coordinate iniziali sono state estratte uniformemente all'interno del doppio intervallo $x \in [-5, 5]$, $y \in [-5, 5]$.

In Figura 8 sono visualizzate sei diverse istanze della routine, tutte convergenti su uno stretto intorno del minimo desiderato: si è impostata una soglia di tolleranza di $\varepsilon = 10^{-15}$ per arrestare la ricerca. In altri casi invece l'algoritmo ha selezionato dei minimi locali disseminati sul plateau, tutti con valori di f nell'ordine di -8×10^{-5} , di cui abbiamo ritrovato riscontro in letteratura. Sei esempi di questo esito sono riportati in Figura 9.

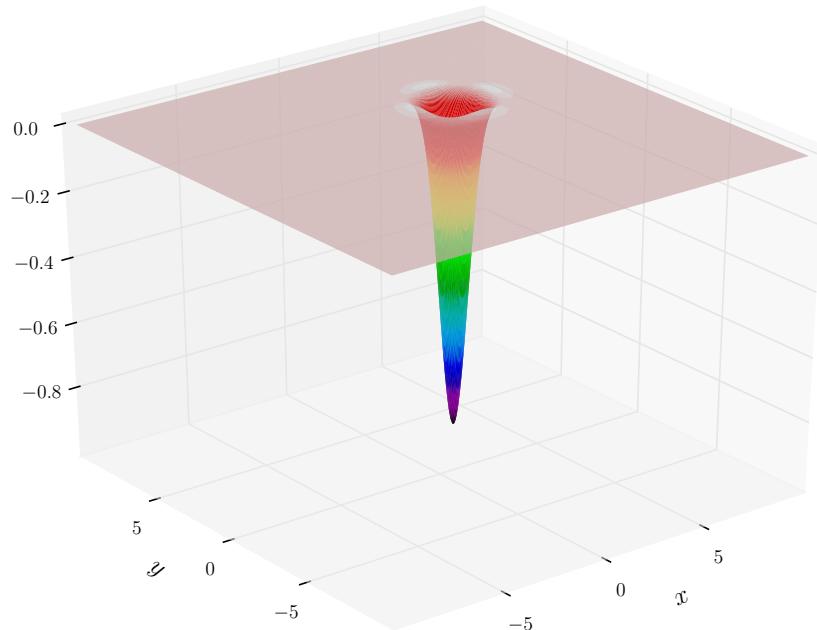


Figura 7: Grafico 3D della funzione di Easom. Da Wikipedia (Cfr. la nota a piè di pagina 25).

¹⁹Ci siamo riferiti a:

- https://en.wikipedia.org/wiki/Test_functions_for_optimization
- <http://benchmarkfcns.xyz/fcns>

Figura 8: Sei istanze in cui il simplesso è andato a convergere sul minimo globale della funzione di Easom.

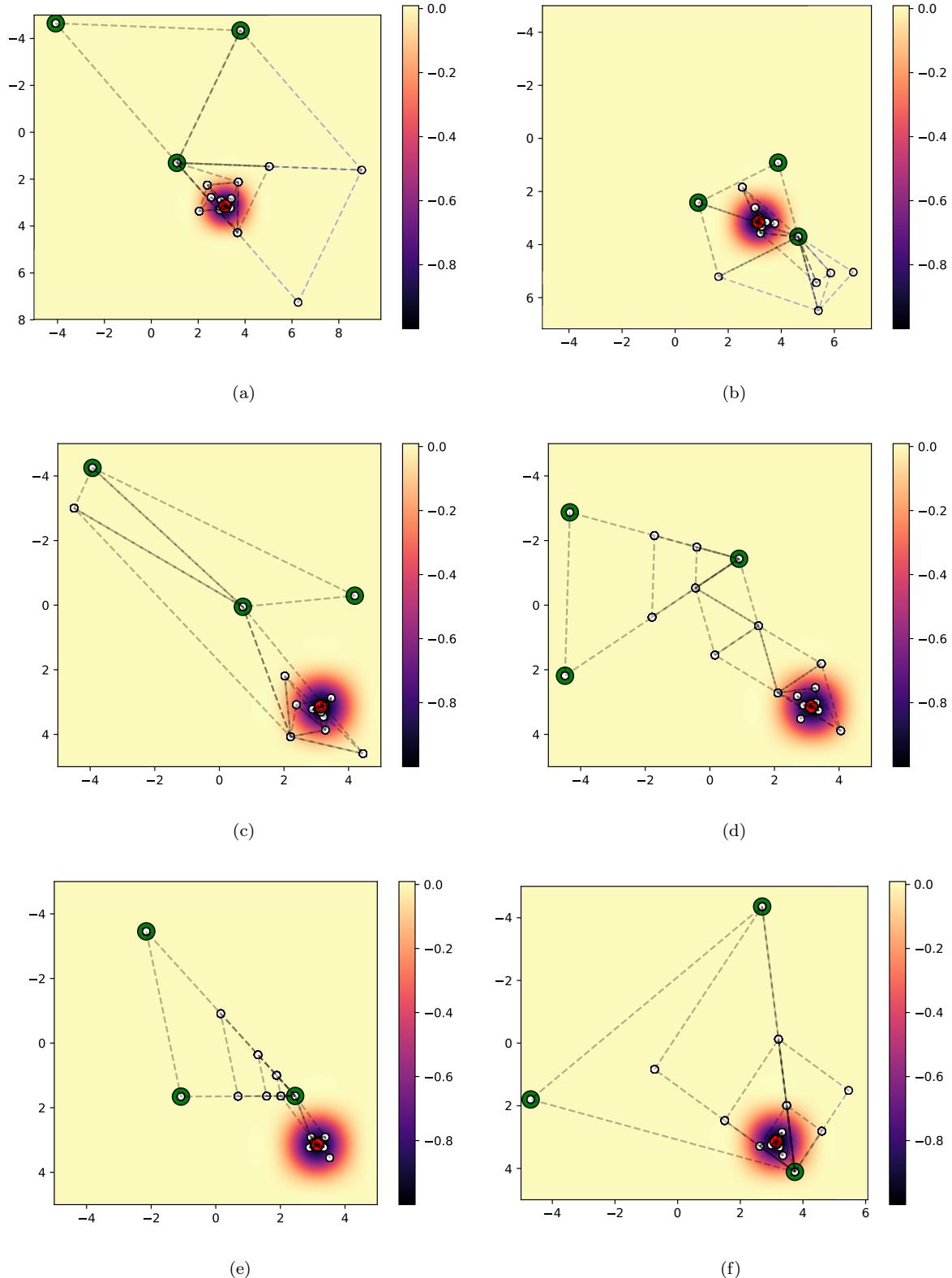
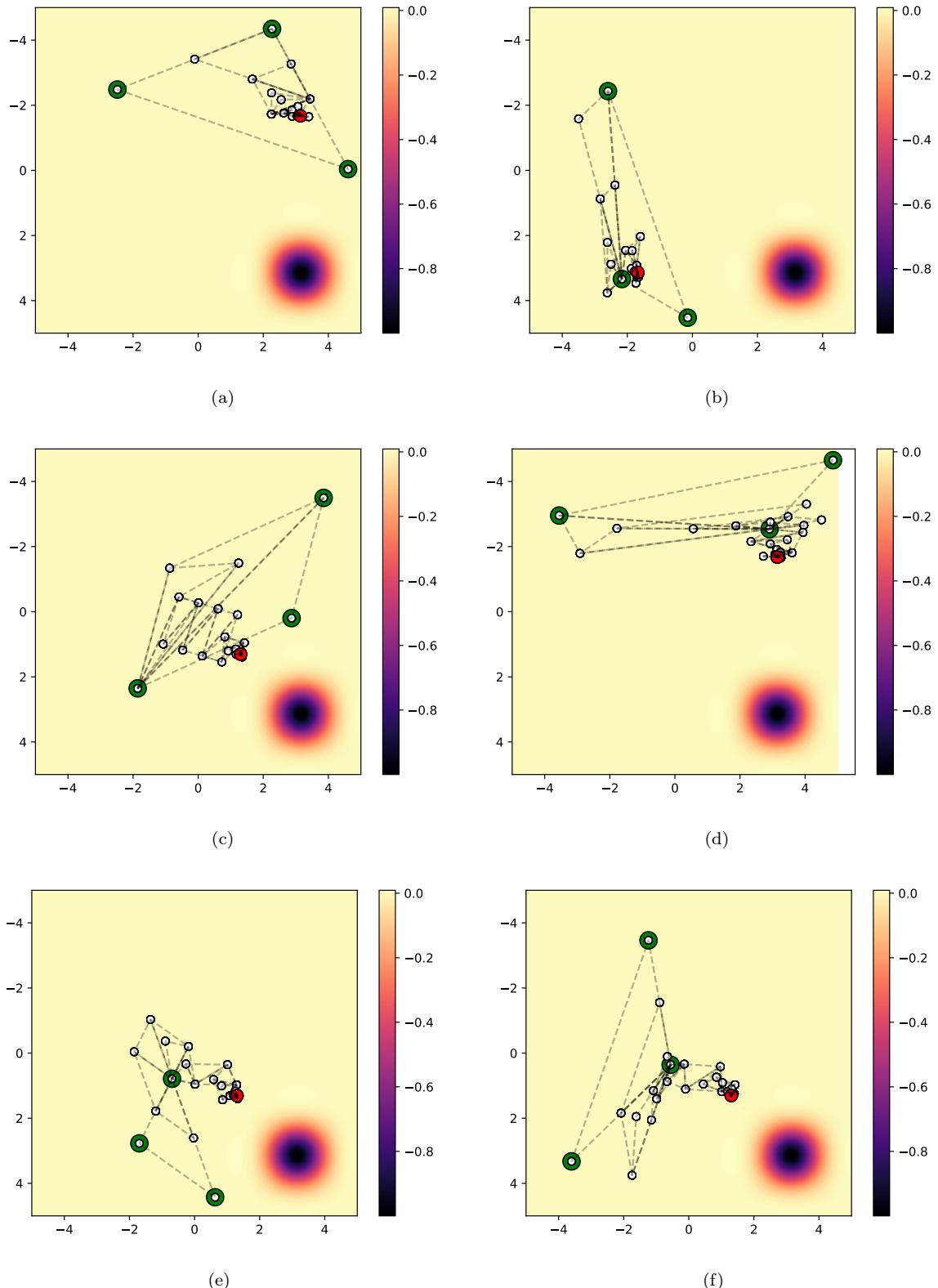


Figura 9: Sei istanze in cui il simplesso ha individuato dei minimi locali sul plateau della funzione di Easom.



Funzione di Goldstein-Price

Anche in questo caso le coordinate iniziali dei vertici sono state generate casualmente nell'intervallo uniforme $[-5, 5]$. Il minimo locale globale è noto essere $f(3, 4) = 1$, ragion per cui tale area di ricerca ci è sembrata senz'altro adeguata.

I grafici in Figura 11 evidenziano chiaramente una *vallata circolare* puntualmente selezionata dall'evoluzione del simplesso. Su di essa effettivamente risiede il minimo globale: si tratta di una circonferenza centrata in $(0, 0)$ e con raggio $R = 5$. In particolare tutte le istanze della routine hanno trovato convergenza in due minimi: quello globale e uno locale nei dintorni di $(5, 0)$.

Tali risultati appaiono del tutto ragionevoli se confrontati con il grafico in scala logaritmica riportato in Figura 10, che evidenzia appunto la struttura della corona circolare individuata dall'evoluzione del simplesso: si tratta di un vallata ad anello con fondo sostanzialmente piatto, tranne che per una "cresta" di minimi locali posizionati circa nel primo quadrante; i due più profondi sembrano essere proprio quelli selezionati dalla nostra routine.

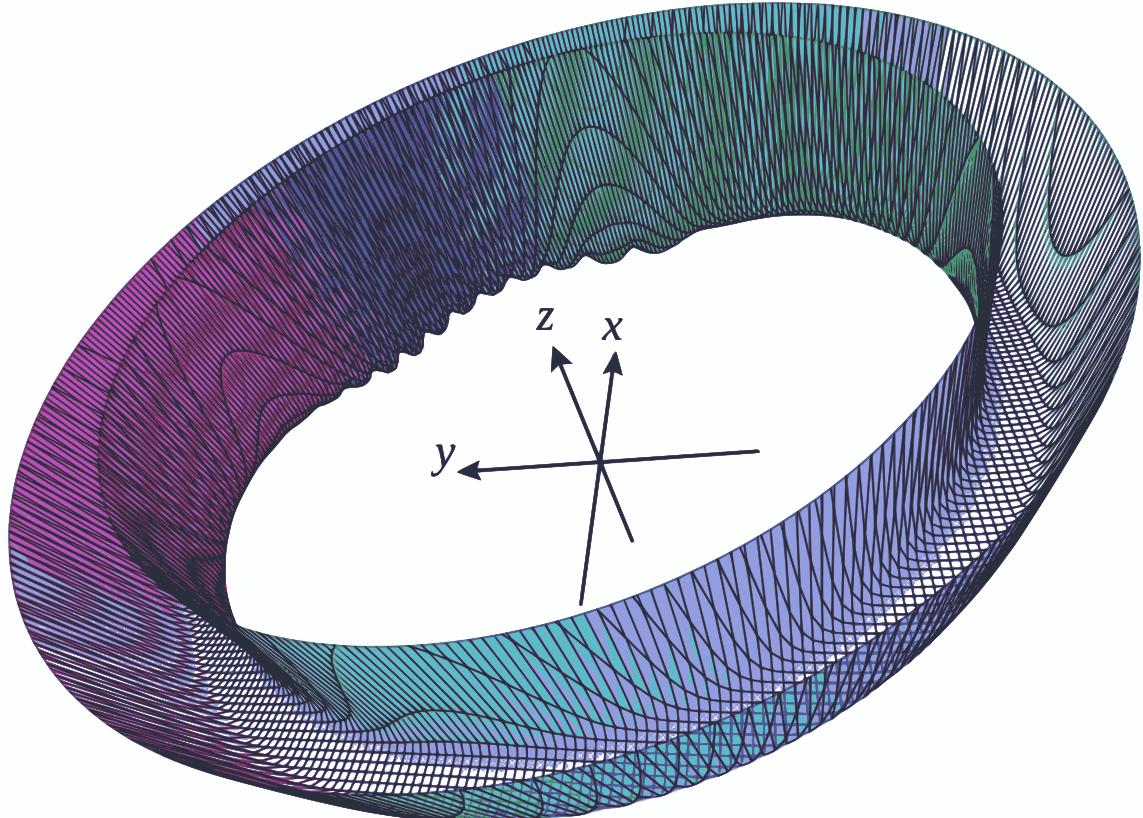
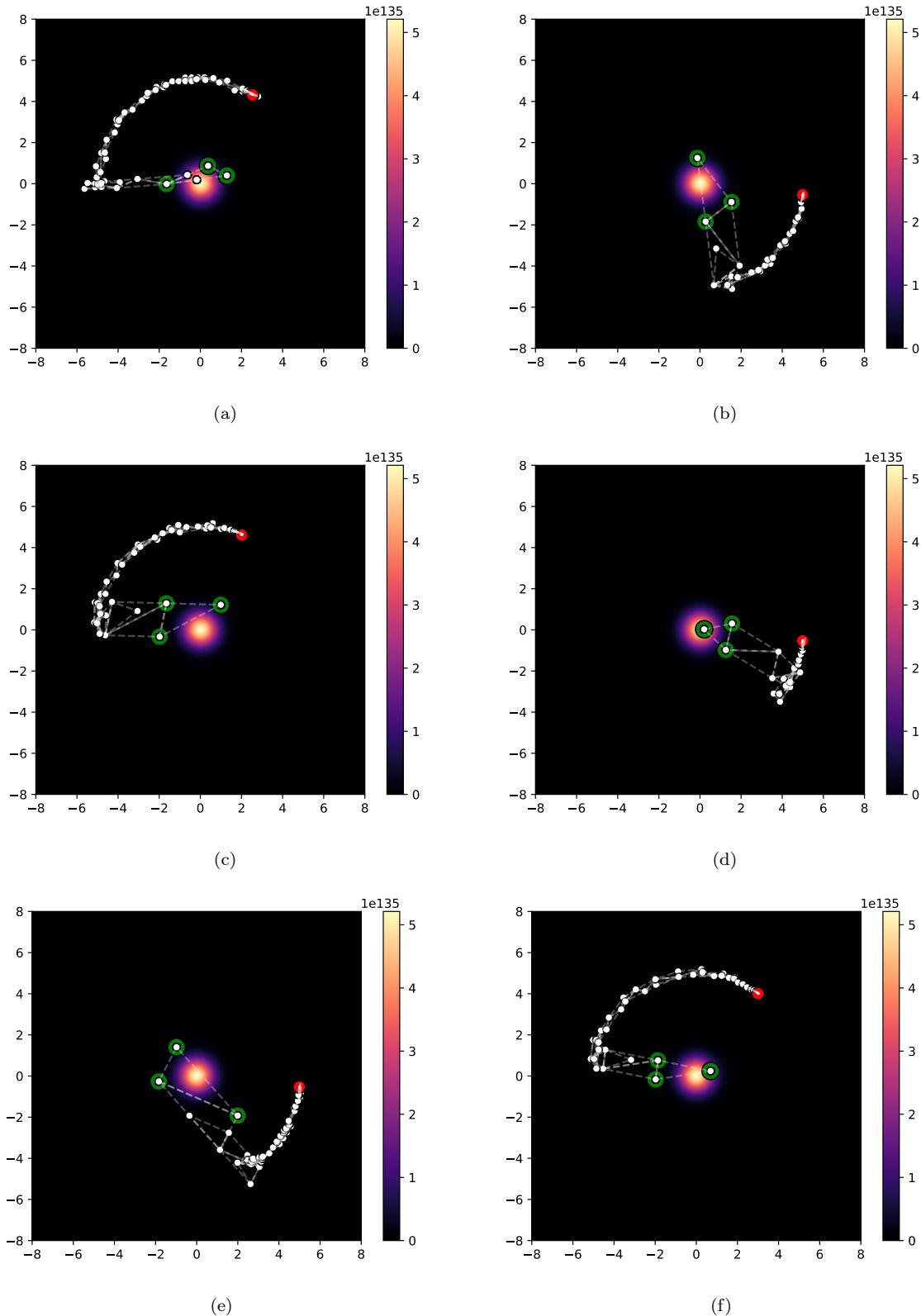


Figura 10: Grafico logaritmico della *vallata circolare* in cui si trova il minimo globale della funzione di Goldstein-Price.

Figura 11: Sei istanze che evidenziano la *vallata circolare* della funzione di Goldstein-Price. In particolare (a), (c) e (f) hanno selezionato il minimo globale $f(3, 4) = 1$. La colormap è in scala logaritmica.



Funzione di Bukin Nr. 6

In Figura 7 è riportato il grafico della funzione Nr.6 di Bukin. È evidente la presenza di una "cresta" di minimi locali posizionati lungo una fascia che interseca perpendicolarmente l'asse \vec{x} nell'origine.

Estraendo coordinate uniformi nella regione asimmetrica $x \in [-5, 5]$, $y \in [-12, 12]$ ne abbiamo individuati vari (Cfr. Figura 13), anche relativamente lontani dal centro della fascia. Ne dedurremmo che i bordi della "cresta" non siano monotoni, anche se rimane il dubbio che in questo caso l'algoritmo implementato soffra di qualche criticità: il simplex è sempre arrivato a convergere su intorni dati da una tolleranza di $\varepsilon = 10^{-5}$ e valori inferiori hanno sempre prodotto tempi di calcolo eccessivamente prolungati.

Per quanto riguarda il minimo globale della funzione, posizionato lontano dalla cresta, in $\mathbf{x}_0 = (-10, 1)$, abbiamo designato una regione stretta e mirata per la generazione dei vertici iniziali del simplexo - doppio intervallo uniforme $x \in [-12, -8]$, $y \in [0, 3]$ - che ci ha permesso di arrivare ad approssimare con buona accuratezza il risultato:

$$\begin{aligned}\mathbf{x}_1 &= (-9.94965119, 0.98995562) &\implies f(\mathbf{x}_1) &= 0.0005071942778401883 \\ \mathbf{x}_2 &= (-9.95151164, 0.99032607) &\implies f(\mathbf{x}_2) &= 0.0005075673516130408 \\ \mathbf{x}_3 &= (-9.95538255, 0.99109576) &\implies f(\mathbf{x}_3) &= 0.0005122552638696298\end{aligned}$$

Il grafico dell'istanza che ha selezionato tali valori è riportato in Figura 14.

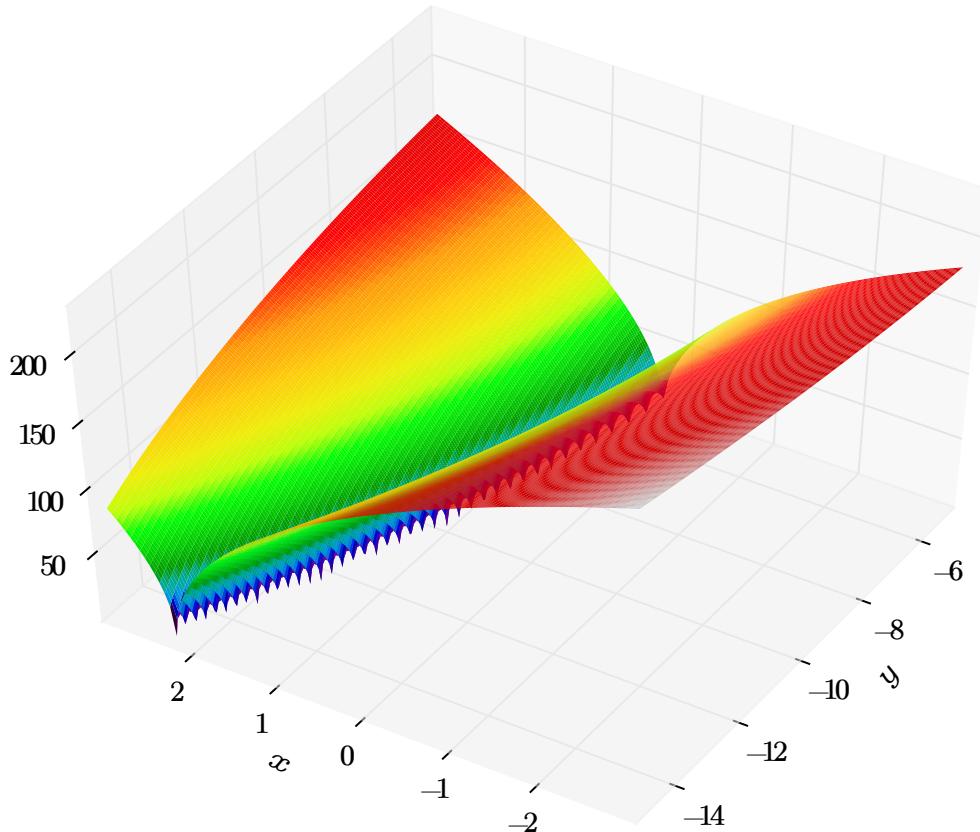


Figura 12: Grafico 3D della funzione di Bukin Nr. 6. Da Wikipedia (Cfr. la nota a piè di pagina 25).

Figura 13: Sei istanze mirate alla ricerca dei minimi locali sulla "cresta" della funzione Nr.6 di Bukin.

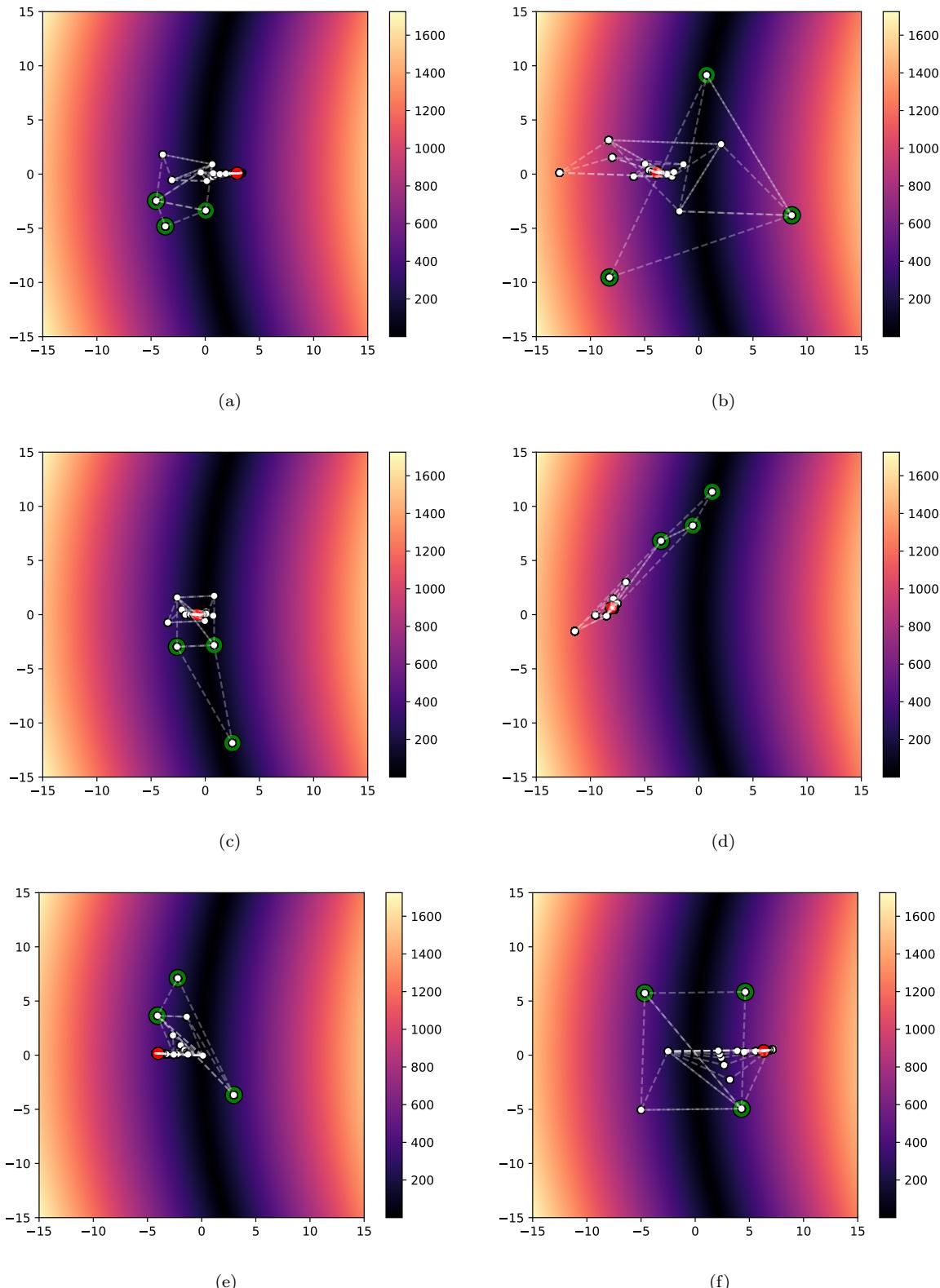
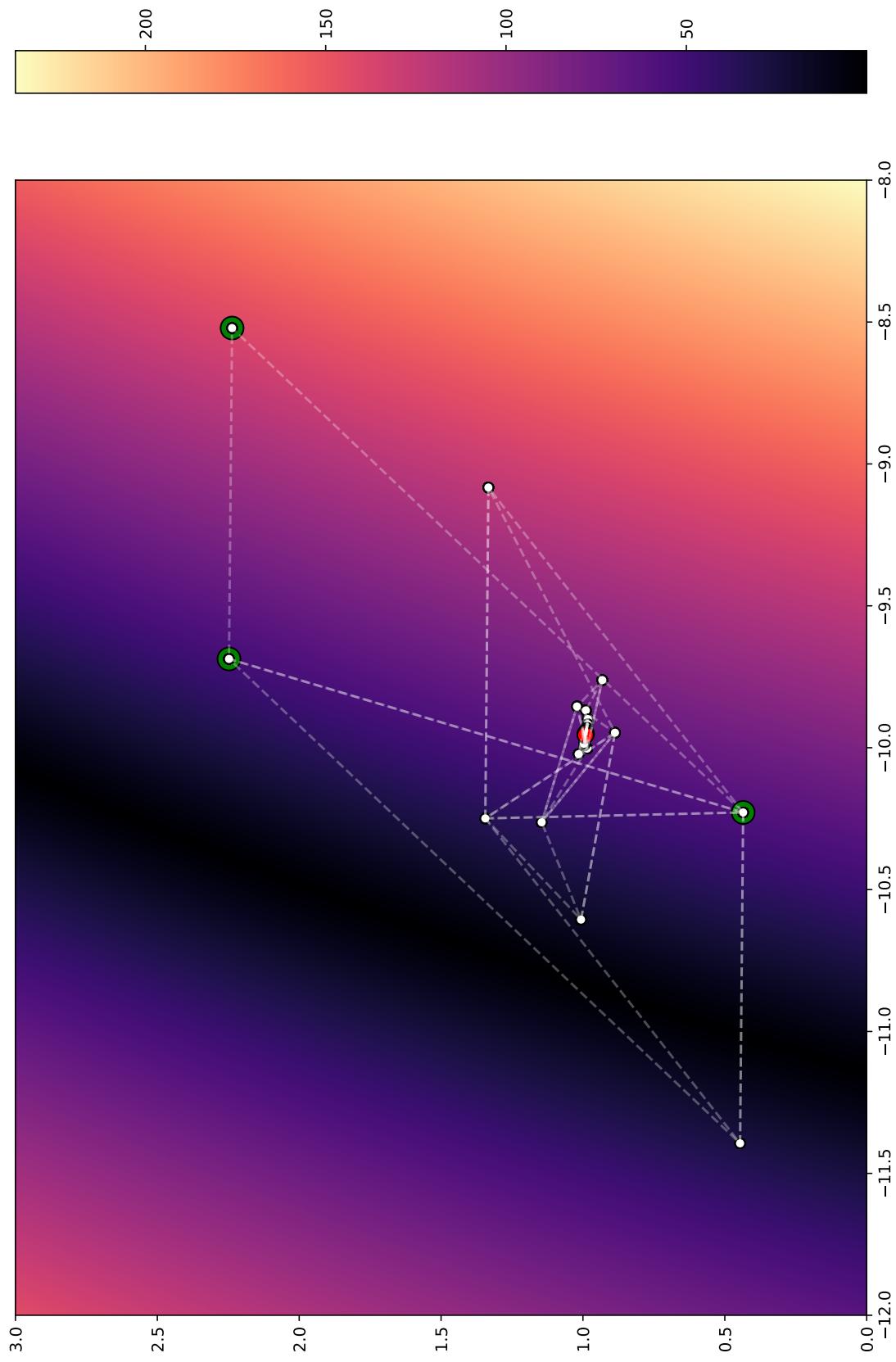


Figura 14: Ricerca del minimo globale della funzione Nr. 6 di Bukin.



Funzione di Booth

La funzione di Booth è unimodale e convessa e in quanto tale presenta un unico minimo globale: $f(1, 3) = 0$. Un suo grafico è riportato in Figura 15.

In questa situazione l'algoritmo di Nelder-Mead è estremamente efficiente e si può generare il simplesso iniziale in una regione vasta a piacere. Noi abbiamo estratto le coordinate dei vertici nel doppio intervallo uniforme $[-15, 15]$.

In Figura 16 sono riportati i grafici relativi a sei istanze della routine, ognuna delle quali ha dato convergenza al punto ottimo (entro $\varepsilon = 10^{-15}$) dopo qualche decina di passi.

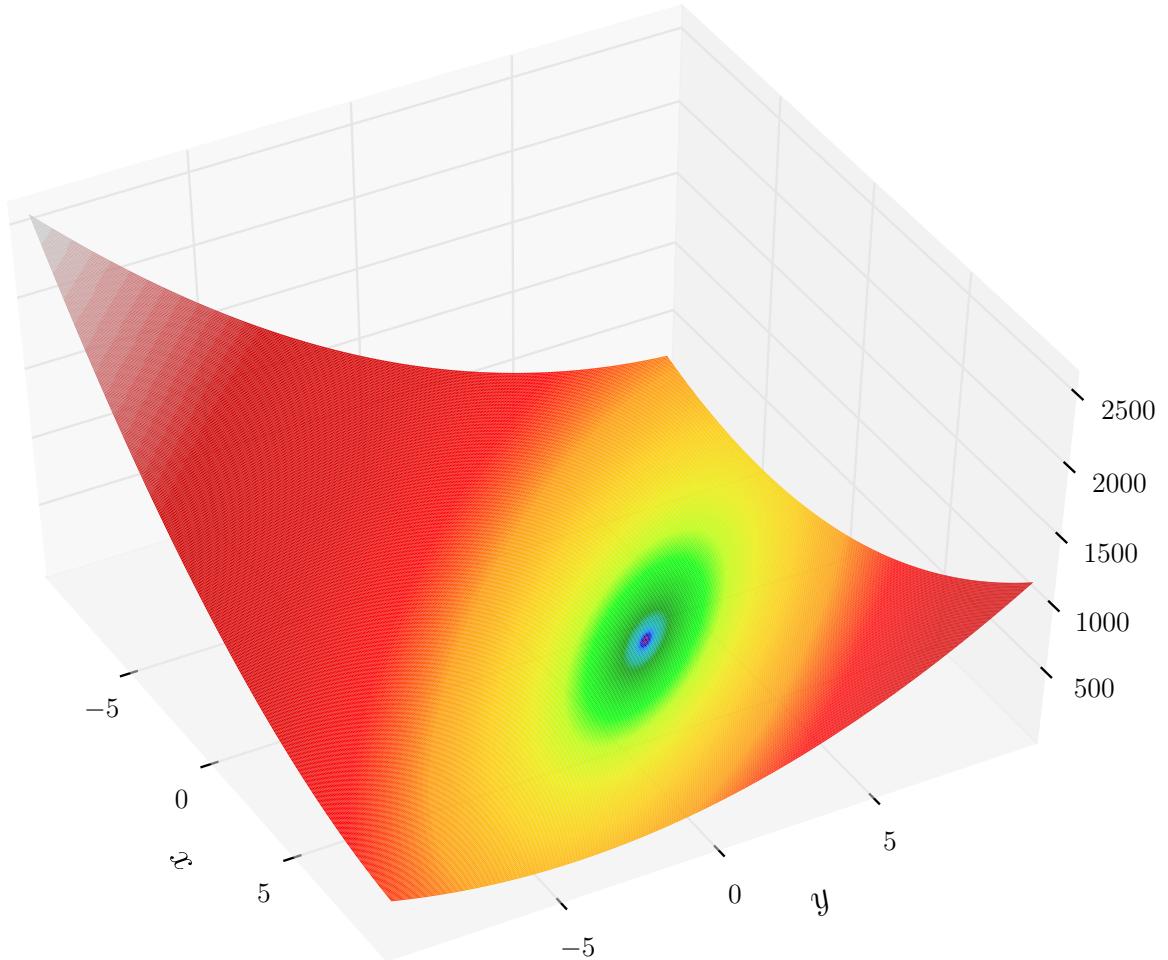
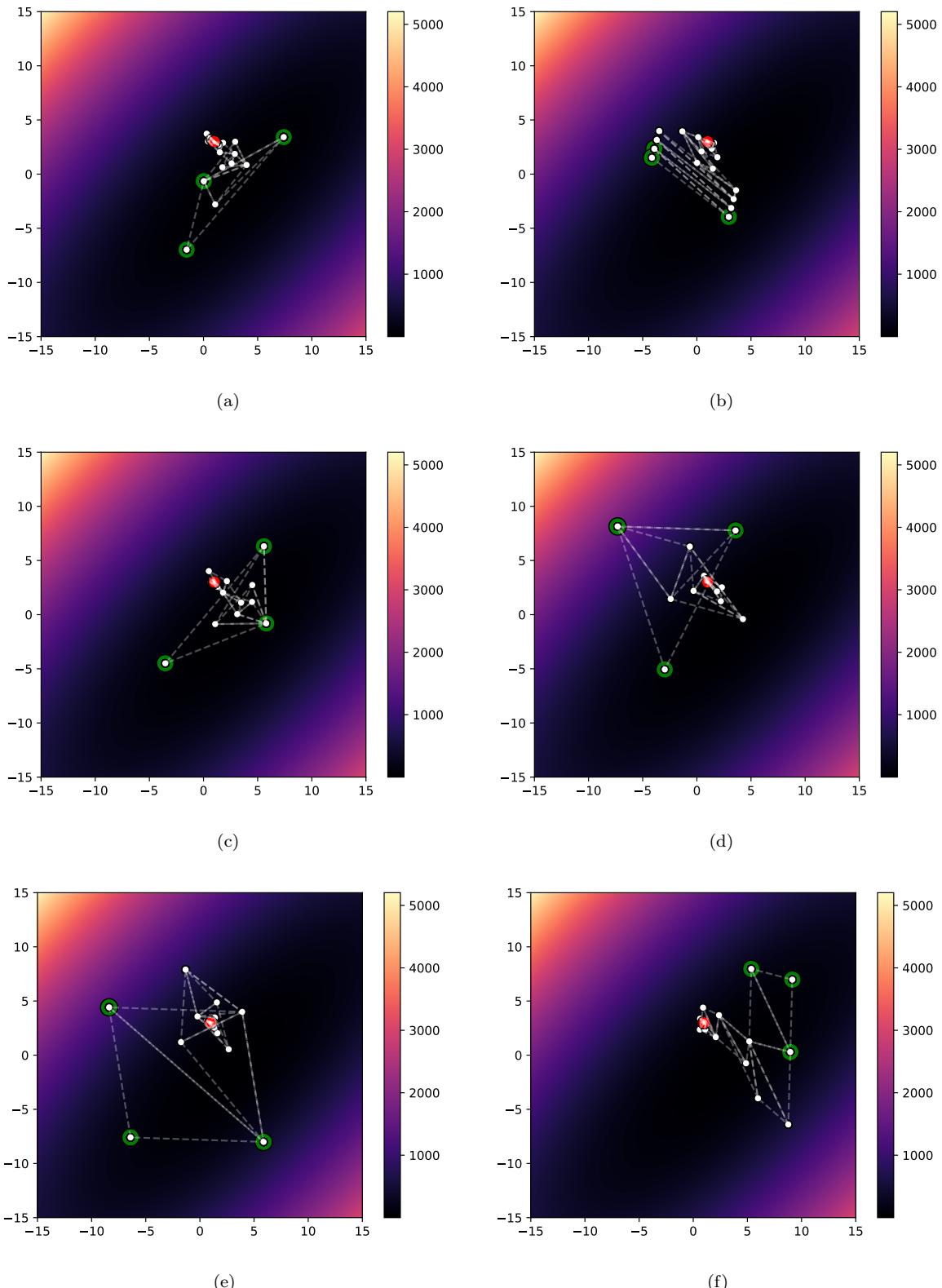


Figura 15: Grafico 3D della funzione di Booth. Da [Wikipedia](#) (Cfr. la nota a piè di pagina 25).

Figura 16: Sei istanze in cui il simplesso è andato a convergere sul minimo globale della funzione di Booth.



Esercizio 4

Si calcoli con un metodo MC l'integrale $\int_{-1}^{+1} (x^5 + x^4) dx$ e si discutano le possibili tecniche di riduzione della varianza.

* * *

Consideriamo il generico integrale $S = \int_a^b f(x) dx$, con $b > a \in \mathbb{R}$ e $f(x)$ funzione integrabile di variabile reale.

Definendo $g(x)$ come la densità di probabilità uniforme nell'intervallo $[a, b]$ possiamo riscrivere l'integrale S come

$$(b - a) \int_{\mathbb{R}} f(x) g(x) dx = (b - a) \int_{\mathbb{R}} \mathbb{I}_{[a,b]} \frac{f(x)}{b - a} dx = \int_a^b f(x) dx = S,$$

e quindi definire un valore medio F tale per cui $S = (b - a)F$:

$$F = \int_{\mathbb{R}} f(x) g(x) dx.$$

Per la *legge dei grandi numeri* (Cfr. Figura 17) possiamo inoltre costruire uno stimatore per F utilizzando un generatore di numeri pseudo-casuali distribuiti secondo $g(x)$. Indicata con $\{x_i\}$ la sequenza generata e con N la sua numerosità scriviamo allora:

$$\hat{F}_N = \frac{1}{N} \sum_{i=1}^N f(x_i), \quad \lim_{N \rightarrow \infty} (b - a)\hat{F}_N = S. \quad (4)$$

Per N finito possiamo anche valutare l'incertezza sulla stima di S per mezzo di uno stimatore della varianza di \hat{F}_N :

$$\hat{S}_N = \sqrt{\frac{1}{N-1} \left(\sum_{i=1}^N [(f(x_i))^2] - [\hat{F}_N]^2 \right)} \quad (5)$$

$$S = (b - a) \left(\hat{F}_N \pm \frac{\hat{S}_N}{\sqrt{N}} \right) \quad (6)$$

Risulta subito evidente che la dipendenza da N dell'incertezza non è particolarmente felice: pur garantendo uno stimatore non biassato nel limite $N \rightarrow \infty$, la convergenza è molto lenta producendo quindi algoritmi sostanzialmente inefficienti. Da tale considerazione nasce dunque l'interesse per algoritmi che riducano la varianza rispetto al metodo definito dalla (4), senza ovviamente produrre bias sul valor medio. Di seguito illustriamo due diverse tecniche di riduzione della varianza: *Importance Sampling* e *Stratified Sampling*.

Importance Sampling

Sia una generica densità di probabilità $p(x) : [a, b] \mapsto \mathbb{R}^+$, $\int_a^b p(x) dx = 1$. Se in $[a, b]$ tale densità si annulla al più in un sottoinsieme di punti di misura nulla possiamo riscrivere l'integrale S come:

$$S = \int_a^b \frac{f(x)}{p(x)} p(x) dx \simeq \frac{1}{N} \sum_{i=1}^N \frac{f(x_i)}{p(x_i)}, \quad \text{dove le } \{x_i\} \text{ sono distribuite secondo } p(x).$$

In questa situazione possiamo scrivere la varianza teorica di S come

$$\sigma^2[S] = \frac{\sigma^2[f/p]}{N}, \quad (7)$$

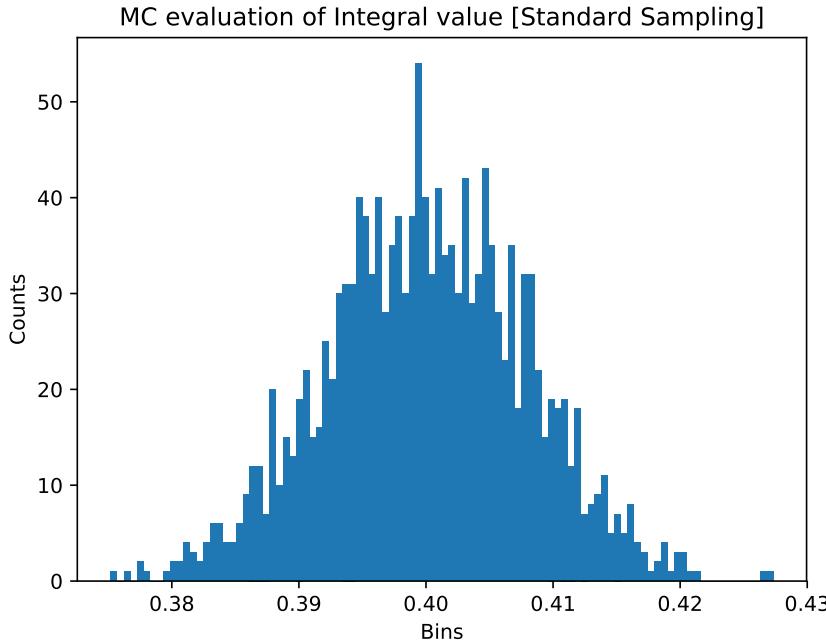


Figura 17: Istogramma relativo alla distribuzione generata per la variabile casuale S nell'implementazione standard del metodo Montecarlo, con $N = 10^4$ eventi generati. È evidente che si tratta in buona approssimazione di una distribuzione normale, in accordo con la legge dei grandi numeri.

e osservare che laddove la densità $p(x)$ risulti proporzionale a $f(x)$ (in $[a, b]!$) la varianza $\sigma^2[f/p] = \sigma^2[\text{const.}]$ risulterà identicamente nulla. Senza pretese di rigore possiamo allora intuire che, fissato N , quanto più " $p(x)$ " sarà simile a " $f(x)$ ", tanto più la varianza di S tenderà a ridursi.

Naturalmente nel caso generale risulta piuttosto difficile generare numeri casuali secondo una distribuzione simile alla funzione desiderata, per cui sostanzialmente si cerca di scrivere l'integrandi come una combinazione lineare di densità note e si calcolano separatamente i rispettivi contributi a S e alla sua varianza. Nel nostro caso si è partizionato l'intervallo $[a, b]$ in maniera tale da poter approssimare la $f(x)$ con la somma di una distribuzione *triangolare* e di una distribuzione *beta*, con supporti disgiunti (Cfr. Figura 18 e Listato 10).

Stratified Sampling

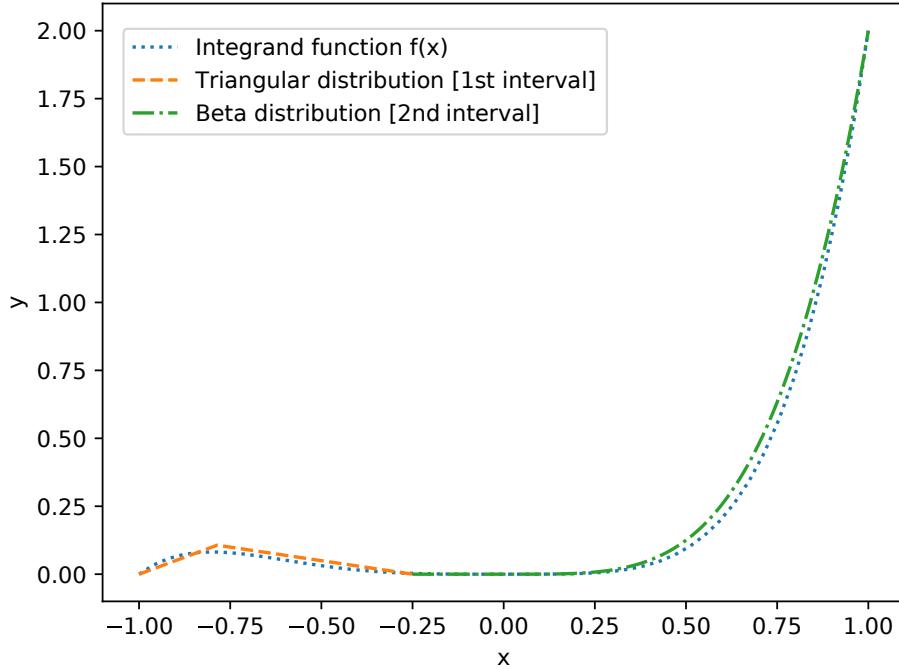
Si tratta di un metodo basato sulla sola proprietà additiva degli integrali: considerato un punto $c \in [a, b]$ si ha analiticamente

$$\int_a^b f(x) dx = \int_a^c f(x) dx + \int_c^b f(x) dx .$$

Tuttavia può essere dimostrato che partizionando l'intervallo $[a, b]$ in sottointervalli sui quali applicare un calcolo Montecarlo "standard" (i.e. con estrazione di numeri casuali distribuiti uniformemente) si ottiene in generale una varianza totale differente rispetto al medesimo calcolo valutato sull'intervallo complessivo, a parità di estrazioni totali. In particolare si può allora ottimizzare la scelta di come ripartire nei sottointervalli il numero di eventi generati, al fine di minimizzare la varianza complessiva del calcolo: in ragione della (7) possiamo argomentare che sarà preferibile campionare maggiormente le regioni in cui la funzione integranda varia più rapidamente, ossia laddove si discosti maggiormente da una densità uniforme. Nel nostro caso (Listato 11) abbiamo definito sette sottointervalli con campionamento crescente da sinistra a destra, per un totale di N eventi generati: in tal modo il calcolo risulta confrontabile con l'implementazione standard (Listato 9).

In linea di massima ci aspettiamo che l'*Importance Sampling* risulti più efficiente nel ridurre la varianza rispetto allo *Stratified Sampling*, a patto che sia possibile approssimare la funzione integranda in combinazioni lineari di densità

Figura 18: Costruzione dell’*Importance Sampling* per il calcolo di S . Le due distribuzioni elementari utilizzate sono state moltiplicate per opportune costanti in modo da evidenziare quanto efficacemente riproducono la funzione data.



di probabilità elementari (i.e. semplici da generare). Nel caso generale invece un campionamento stratificato resta senz’altro l’opzione più flessibile e immediata da implementare.

* * *

Di seguito riportiamo i risultati ottenuti con il calcolo *Standard* (Listato 9), con *Importance Sampling* (Listato 10) e con *Stratified Sampling* (Listato 11), in termini di media e deviazione standard delle gaussiane generate per la variabile aleatoria S . In tutti i casi sono stati estratti complessivamente 10^4 campioni.

$S = \int_{-1}^{+1} (x^5 + x^4) dx$	Media Campionaria	Deviazione Standard Campionaria
<i>Standard Sampling</i>	0.4032560803645552	$8.077310205890864 \times 10^{-3}$
<i>Importance Sampling</i>	0.3989493411116131	$1.113214189831381 \times 10^{-3}$
<i>Stratified Sampling</i>	0.4019606513273493	$1.462526565299890 \times 10^{-3}$

Risulta innanzitutto evidente che i valori medi delle tre stime approssimano bene il valore analitico $S = 0.4$. Inoltre risultano tra loro compatibili (entro meno di 2σ), a riprova del fatto che le tecniche di riduzione della varianza utilizzate non traslano il valor medio dell’integrale. Osserviamo infine come la varianza campionaria diminuisce di un ordine di grandezza passando da *Standard Sampling* a *Stratified Sampling* e di quasi due ordini di grandezza utilizzando l’*Importance Sampling*. Tali risultati soddisfano appieno le nostre aspettative.

Listing 9: Python code for Standard Montecarlo Calculation

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 import math
4
5 ## Integrand function
6 def f(x): return x**5+x**4
7
8 ## Integration interval
9 a = -1.0
10 b = 1.0
11
12 ## Number of random number generations
13 n = 10000
14
15 ## Standard MC implementation
16 h=[]
17 for k in range(1,1500):
18
19     x=np.random.uniform(a,b,n) # [a,b]=[-1.0,1.0]
20     eval_funct=f(x)
21     h.append((b-a)*np.sum(eval_funct)/(n))
22
23 S=(b-a)*(np.sum(eval_funct))/n
24 n=10000.0
25 mu_camp=(np.sum(eval_funct))/n
26 var_camp=1/(n-1)*np.sum((eval_funct-mu_camp)**2)
27 var=(b-a)**2*(1/n)*var_camp
28
29 print (S,'Integral Mean with Standard MC')
30 print (var,'Variance with Standard MC')
31 print(sqrt(var), 'Standard Deviation with Standard MC')
32
33 ## Plotting a histogram of the generated gaussian
34 hist,bin_edges=np.histogram(h,bins=100)
35 plt.figure()
36 plt.hist(h,bin_edges)
37 plt.xlabel('Bins')
38 plt.ylabel('Counts')
39 plt.title('MC evaluation of Integral value [Standard Sampling]')
40 axes=plt.gca()
41
42 plt.show()

```

Listing 10: Python code for Importance Sampling Montecarlo Calculation

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 import math
4 from scipy.stats import triang
5 from scipy.stats import beta
6
7 ## Integrand function
8 def f(x): return x**5+x**4
9
10 x=np.linspace(-1,1,endpoint=True,dtype=float)
11 plt.plot(x,f(x),'.')
12
13 ## 1st Interval: Triangular-approximation

```

```

14 s=np.linspace(-1,-0.25,50)
15 plt.plot(s,0.04*triang.pdf(s,0.2/0.7,a,b-a), '---')
16 a=-1
17 b=-0.25
18 n1=50
19 x1=np.random.triangular(a,-0.8,b,n1)
20 eval_funct1=f(x1)
21 y1=triang.pdf(x1,0.2/0.7,a,b-a)
22 n1=n1*1.0
23
24 i1=(1/n1)*np.sum(eval_funct1/y1)
25
26 mu_camp1=(np.sum(eval_funct1/y1))/n1
27 var_camp1=1/(n1-1)*np.sum((eval_funct1/y1-mu_camp1)**2)
28 var1=(1/n1)*var_camp1
29
30 ## 2nd Interval: Beta-approximation
31 k=np.linspace(-0.25,1,100)
32 plt.plot(k,0.4*beta.pdf(k,5,1), '-.')
33
34 n2=9950
35 x2=beta.rvs(5,1,size=n2)
36 eval_funct2=f(x2)
37 y2=beta.pdf(x2,5,1)
38
39 n2=n2*1.0
40 i2=(1/n2)*np.sum(eval_funct2/y2)
41
42 mu_camp2=(np.sum(eval_funct2/y2))/n2
43 var_camp2=1/(n2-1)*np.sum((eval_funct2/y2-mu_camp2)**2)
44 var2=(1/n2)*var_camp2
45
46 ## Results
47 print(i1+i2,'Integral Mean with Importance Sampling')
48 print(var1+var2,'Total Variance with Importance Sampling')
49 print(sqrt(var1+var2), 'Total Standard Deviation with Importance Sampling')
50 plt.xlabel('x')
51 plt.ylabel('y')
52 plt.show()

```

Listing 11: Python code for Stratified Sampling Montecarlo Calculation

```

1 import numpy as np
2 import math
3
4 ## Integrand function
5 def f(x): return x**5+x**4
6
7 ## 1st Interval
8 a=-1.0
9 b=-0.4
10 n1=100
11 x1=np.random.uniform(a,b,n1) # [a,b]=[-1.0,-0.4]
12 k=np.linspace(0.35,1,50)
13 eval_funct1=f(x1)
14 n1=n1*1.0
15 i1=(b-a)*(np.sum(eval_funct1)/n1) # mean
16 mu_camp1=(np.sum(eval_funct1))/n1
17 var_camp1=(1/(n1-1))*np.sum((eval_funct1-mu_camp1)**2) # variance

```

```

18 var1=(b-a)**2*(1/n1)*var_camp1
19
20 ## 2nd Interval
21 a=-0.4
22 b=0.4
23 n2=100
24 x2=np.random.uniform(a,b,n2) # [a,b]=[-0.4,0.4]
25 eval_func2=f(x2)
26 n2=n2*1.0
27 i2=(b-a)*(np.sum(eval_func2)/n2) # mean
28 mu_camp2=(np.sum(eval_func2))/n2
29 var_camp2=1/(n2-1)*np.sum((eval_func2-mu_camp2)**2) # variance
30 var2=(b-a)**2*(1/n2)*var_camp2
31
32 ## 3rd Interval
33 a=0.4
34 b=0.6
35 n3=800
36 x3=np.random.uniform(a,b,n3) # [a,b]=[0.4,0.6]
37 eval_func3=f(x3)
38 n3=n3*1.0
39 i3=(b-a)*(np.sum(eval_func3)/n3) # mean
40 mu_camp3=(np.sum(eval_func3))/n3
41 var_camp3=1/(n3-1)*np.sum((eval_func3-mu_camp3)**2) # variance
42 var3=(b-a)**2*(1/n3)*var_camp3
43
44 ## 4th Interval
45 a=0.6
46 b=0.7
47 n4=1000
48 x4=np.random.uniform(a,b,n4) # [a,b]=[0.6,0.7]
49 eval_func4=f(x4)
50 n4=n4*1.0
51 i4=(b-a)*(np.sum(eval_func4)/n4) # mean
52 mu_camp4=(np.sum(eval_func4))/n4
53 var_camp4=1/(n4-1)*np.sum((eval_func4-mu_camp4)**2) # variance
54 var4=(b-a)**2*(1/n4)*var_camp4
55
56 ## 5th Interval
57 a=0.7
58 b=0.8
59 n5=1500
60 x5=np.random.uniform(a,b,n5) # [a,b]=[0.7,0.8]
61 eval_func5=f(x5)
62 n5=n5*1.0
63 i5=(b-a)*(np.sum(eval_func5)/n5) # mean
64 mu_camp5=(np.sum(eval_func5))/n5
65 var_camp5=1/(n5-1)*np.sum((eval_func5-mu_camp5)**2) # variance
66 var5=(b-a)**2*(1/n5)*var_camp5
67
68 ## 6th Interval
69 a=0.8
70 b=0.9
71 n6=3000
72 x6=np.random.uniform(a,b,n6) # [a,b]=[0.8,0.9]
73 eval_func6=f(x6)
74 n6=n6*1.0
75 i6=(b-a)*(np.sum(eval_func6)/n6) # mean
76 mu_camp6=(np.sum(eval_func6))/n6

```

```

77 var_camp6=1/(n6-1)*np.sum((eval_funct6-mu_camp6)**2) # variance
78 var6=(b-a)**2*(1/n6)*var_camp6
79
80 ## 7th Interval
81 a=0.9
82 b=1.0
83 n7=3500
84 x7=np.random.uniform(a,b,n7) # [a,b]=[0.9,1]
85 eval_funct7=f(x7)
86 n7=n7*1.0
87 i7=(b-a)*(np.sum(eval_funct7)/n7) # mean
88 mu_camp7=(np.sum(eval_funct7))/n7
89 var_camp7=1/(n7-1)*np.sum((eval_funct7-mu_camp7)**2) # variance
90 var7=(b-a)**2*(1/n7)*var_camp7
91
92 ## Results
93 print(n1+n2+n3+n4+n5+n6+n7,'Check on total number of generations [has to be 10^4]')
94 S=i1+i2+i3+i4+i5+i6+i7
95 print(S,'Total mean for S, with Stratified Sampling')
96 var=var1+var2+var3+var4+var5+var6+var7
97 print(var,'Total variance for S, with Stratified Sampling')
98 dev = sqrt(var)
99 print(dev,'Total standard deviation for S, with Stratified Sampling')

```

Esercizio 5

- A. Nel dropbox del corso sono presenti alcuni files di dati del tipo `CeBr3_xx.dat` con `xx` nome di una sorgente di calibrazione: Cs^{137} , Co^{57} , Na^{22} , Ba^{133} e Eu^{152} .
I dati sono lo spettro in formato ASCII di un MCA (con 8K canali) per un cristallo di CeBr_3 esposto a varie sorgenti di calibrazione. Istruire i vari spettri di multicanale e scrivere un programma per determinare posizione e risoluzione spettrale dei picchi di calibrazione, il più automatizzato possibile.
Si discutano i risultati ottenuti in termini di risoluzione energetica del cristallo e linearità.
- B. Utilizzando gli algoritmi di smoothing illustrati a lezione, provare a vedere se si riesce a migliorare la determinazione spettrale dei picchi spettrali trovati: area, altezza... Discutere i risultati ottenuti.

* * *

In Figura 19 sono riportati i grafici dei dati, così come riportati nei file di input relativi all'esposizione del cristallo alle diverse sorgenti (conteggi "grezzi"). Per individuarne in maniera automatizzata i picchi di risonanza si è pensato innanzitutto di dividere il dominio spettrale dell'analizzatore multicanale in sotto-bande sulle quali determinare il massimo assoluto dei conteggi: scegliendo opportunamente la larghezza di queste regioni di ottimizzazione locale si ha buona speranza di individuare un sottoinsieme di canali che includa tutti i bin contenenti le frequenze di risonanza cercate. Naturalmente alcuni dei massimi così individuati saranno solo artefatti della suddivisione operata sulla banda del rivelatore (ad esempio valori al bordo di una sotto-banda in cui lo spettro è monotono) per cui rigettiamo tutti quelli che non soddisfino le condizioni elementari che definiscono un punto stazionario di massimo:

1. derivata prima dei conteggi compresa in un piccolo intorno dello zero;
2. derivata seconda dei conteggi minore di una certa soglia negativa.

A questo punto si saranno certamente individuati dei massimi locali dell'intero spettro che siano anche ottimi globali nelle rispettive sotto-regioni di ricerca.

Resta tuttavia qualche criticità su eventuali *plateau* degli spettri analizzati: mancando nelle sotto-regioni interessate dei picchi "veri", le oscillazioni riconducibili al rumore intrinseco della rivelazione danno inevitabilmente luogo a *falsi positivi* (ossia massimi locali non associati a risonanze del cristallo scintillatore). Per ovviare a tal problema è stata definita un'ulteriore condizione necessaria per l'accettazione dei valori selezionati:

3. numero di conteggi maggiore di una certa soglia, associabile ai picchi di rumore.

Naturalmente tutta la difficoltà di implementazione dell'algoritmo risiede nella definizione dei parametri per le condizioni (1), (2) e (3), considerate soprattutto la richiesta di massima automazione della routine e la conseguente sconvenienza di un'accordatura manuale da operare di volta in volta sulla base dei dati analizzati. Si è scelto di far dipendere opportunamente tali parametri dalla sotto-banda di origine del candidato picco e in particolare di considerare:

- « `a = 1.10 * mean{Counts on Sub-Region}` » come soglia sui conteggi ($y > a$);
- « `b = 0.45 * max{Derivative - Counts on Sub-Region}` » come soglia sulla pendenza ($|y'| < b$);
- « `c = 0.05 * max{2ndDerivative - Counts on Sub-Region}` » come soglia sulla concavità ($y'' < -c$).

Le derivate dei conteggi sono state valutate numericamente per mezzo di differenze finite al secondo ordine, utilizzando la libreria `numpy.gradient` di Python²⁰.

Infine, per assicurarsi di eliminare qualsiasi artefatto ancora dovuto al particolare partizionamento dello spettro, si è pensato di ripetere l'intera procedura con una differente scelta per la larghezza dei sotto-intervalli e di accettare

²⁰Documentazione consultabile al seguente link: <https://docs.scipy.org/doc/.../numpy.gradient.html>

definitivamente soltanto i candidati picchi individuati da entrambe le istanze della routine. In particolare, definito innanzitutto il range effettivo dei dati considerati (tutti gli spettri hanno una soglia energetica oltre la quale sono sostanzialmente nulli), l'intervallo risultante è stato suddiviso la prima volta in 20 e la seconda in 8 parti uguali, sulle quali sono stati definiti i candidati picchi, poi filtrati secondo le condizioni (1), (2) e (3). Il confronto diretto tra i due *data-set* così ottenuti ha decretato quali candidati accettare come effettivi picchi di risonanza.

In Figura 20 sono indicati (*croci arancioni*) i punti che l'algoritmo proposto ha individuato sugli spettri: risulta subito evidente che qualche picco importante non è stato individuato, soprattutto su Co⁵⁷ e Na²². D'altra parte ci riteniamo soddisfatti dal grado di selettività della routine rispetto ai falsi positivi, considerato l'alto livello di rumore presentato da alcuni degli spettri proposti. I dettagli dell'implementazione in Python possono essere consultati nella prima parte del Listato 12.

Per indagare la risoluzione energetica del cristallo scintillatore abbiamo deciso di analizzare per ogni spettro il Picco Principale (P.P.), inteso come quello relativo alla risonanza che ha prodotto più conteggi sul rivelatore: la strategia è mirata alla minimizzazione dei problemi dovuti al rumore delle misure.

Innanzitutto è stato necessario definire una ROI (Region Of Interest) attorno al P.P. in modo da poter operare un *fitting* parametrico dei conteggi relativi alla sola risonanza considerata. Si è assunto che il picco abbia profilo simil-Gaussiano, al netto di un eventuale fondo da sottrarre opportunamente. In particolare la ROI attorno al picco è stata definita come il minimo intorno del punto di massimo tale da soddisfare le seguenti condizioni, scelte in modo da funzionare robustamente per tutti gli spettri proposti:

$$|y_{\text{ROI}} - y_{\max}| \leq \frac{3}{4} \times y_{\max},$$

$$|y'_{\text{ROI}}| < 50.$$

Il fondo è stato approssimato al segmento congiungente i due punti spettrali agli estremi della ROI, per mezzo della libreria Python adibita all'interpolazione polinomiale con metodo l.s. (*least squares*): `numpy.polyfit`²¹. Infine anche il fit Gaussiano è stato ottenuto con metodo l.s., ma adoperando la più generica libreria `scipy.optimize.curve_fit`²² che permette di definire a piacere il modello da adattare ai dati.

A partire dunque dalla definizione:

$$g(x) = A \exp\left[\frac{-(x - x_0)^2}{2\sigma^2}\right],$$

e fornendo al *fitter* i parametri iniziali:

$$A \Big|^{(\text{init})} = y_{\max}, \quad x_0 \Big|^{(\text{init})} = x_{\max}, \quad \sigma \Big|^{(\text{init})} = \hat{\sigma}_x(\text{DATA}),$$

otteniamo i risultati²³ riportati nelle Figure 21, 22, 23, 24 e 25.

²¹Documentazione: <https://docs.scipy.org/doc/.../numpy.polyfit.html>

²²Documentazione: https://docs.scipy.org/doc/.../scipy.optimize.curve_fit.html

²³Larghezza del picco w e relativa risoluzione spettrale w/x_0 sono calcolate come da standard FWHM. L'area sottesa è stimata in approssimazione triangolare $S \simeq wA/2$. Le incertezze infine sono ricavate dalla matrice di covarianza restituita dal *fitter*.

Figura 19: Istogrammi grezzi dei cinque spettri di calibrazione considerati.

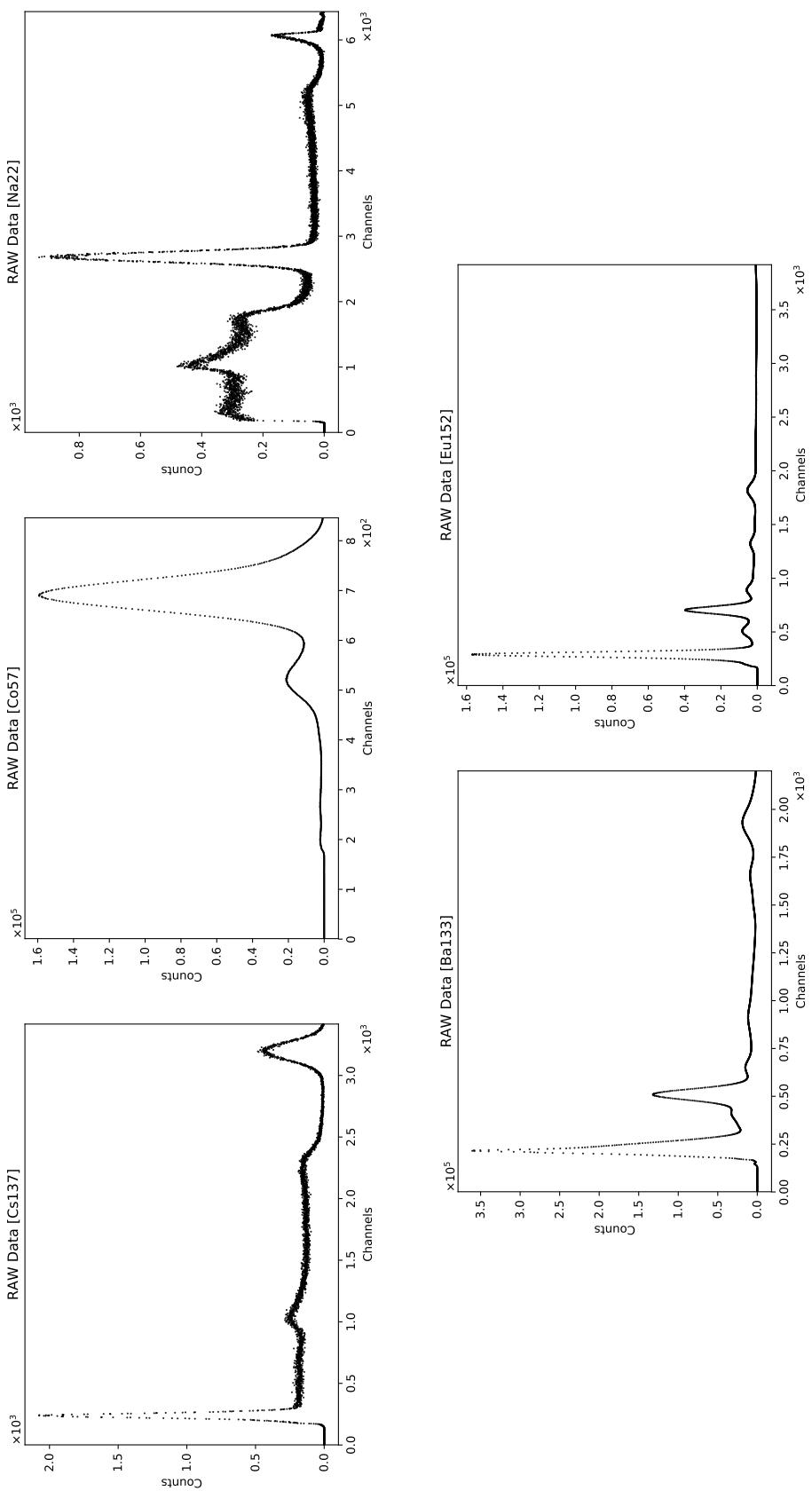
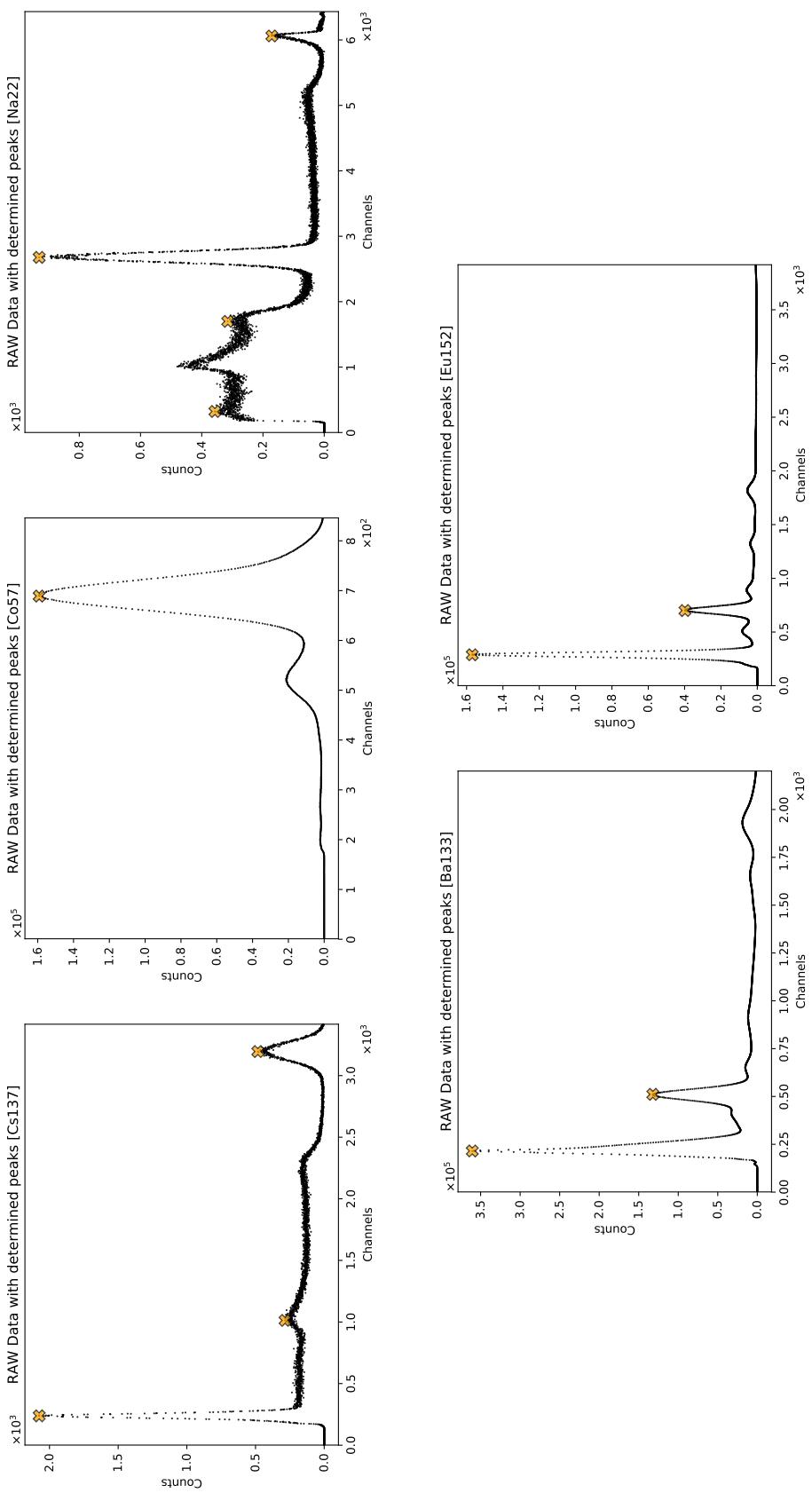


Figura 20: Ricerca dei picchi di scintillazione sugli spettri di calibrazione grezzi.



Listing 12: Python code for Spectral Peak Recognition Routine

```

1  """ Standard Libraries """
2  import numpy as np
3  import pylab as plt
4  from math import *
5
6  """ I/O Libraries """
7  import os
8  import sys
9
10 """ Best-Fit Library """
11 from scipy.optimize import curve_fit
12
13 ######
14
15 ## Choosing the source
16
17 SourceName = str(input('Insert the desired source [Cs137|Co57|Na22|Ba133|Eu152]: '))
18
19 ## Opening the ASCII file related to the chosen source
20
21 def get_script_path():
22     return os.path.dirname(os.path.realpath(sys.argv[0]))
23
24 path = get_script_path()
25 data=np.loadtxt(path+'/Script&Data/CeBr3_'+SourceName+'.dat')
26
27 ## Defining Effective-Range
28
29 i = len(data)
30 while(data[i-1] <= 0.01*np.max(data)/2):
31     i = i - 1
32
33 x_max = i
34 data = data[range(1,x_max)]
35
36 ## Plotting retrieved experimental data
37
38 x = np.array(range(1,x_max))
39 y = data
40
41 plt.figure('Raw'+SourceName)
42 plt.plot(x,y,'ko',markersize=0.5)
43 plt.xlim((1, x_max))
44 plt.xlabel('Channels')
45 bottom, top = plt.ylim()
46 plt.ylabel('Counts')
47 plt.title('RAW Data ['+SourceName+']')
48 plt.ticklabel_format(axis='both',style='sci',scilimits=(0,0),useMathText=True)
49 plt.show()
50
51 #####
52 ## Peak-Recognition Routine ##
53 #####
54
55 # Derivatives of data
56 y1 = np.gradient(y)
57 y2 = np.gradient(y1)

```

```

58
59 X_Peaks1 = np.array([])
60 Y_Peaks1 = np.array([])
61 X_Peaks2 = np.array([])
62 Y_Peaks2 = np.array([])
63 Y1 = np.array([])
64 Y2 = np.array([])
65
66 steps = 20
67 for i in range(1,steps+1):
68
69 # Minimization only in a partial domain
70 X = x[(x > (x_max/steps)*(i-1)) & (x <= (x_max/steps)*i)]
71 Y = y[(x > (x_max/steps)*(i-1)) & (x <= (x_max/steps)*i)]
72 Y1 = y1[(x > (x_max/steps)*(i-1)) & (x <= (x_max/steps)*i)]
73 Y2 = y2[(x > (x_max/steps)*(i-1)) & (x <= (x_max/steps)*i)]
74 X_CandidatePeak = X[np.argmax(Y)]
75 Y_CandidatePeak = np.max(Y)
76 Y1_CandidatePeak = Y1[np.argmax(Y)]
77 Y2_CandidatePeak = Y2[np.argmax(Y)]
78
79 a = 1.1*np.max(Y)/2           # Threshold on Background
80 b = 0.9*np.max(abs(Y1))/2    # Threshold on Slope
81 c = 0.1*np.max(abs(Y2))/2    # Threshold on Concavity
82
83 if (Y_CandidatePeak > a)and(abs(Y1_CandidatePeak)< b)and(Y2_CandidatePeak<-c)and(
84     Y_CandidatePeak > 0.13*np.max(y)):
85
86     X_Peaks1 = np.append(X_Peaks1, X_CandidatePeak)
87     Y_Peaks1 = np.append(Y_Peaks1, Y_CandidatePeak)
88
89 steps = 8
90 for i in range(1,steps+1):
91
92 # Minimization only in a partial domain
93 X = x[(x > (x_max/steps)*(i-1)) & (x <= (x_max/steps)*i)]
94 Y = y[(x > (x_max/steps)*(i-1)) & (x <= (x_max/steps)*i)]
95 Y1 = y1[(x > (x_max/steps)*(i-1)) & (x <= (x_max/steps)*i)]
96 Y2 = y2[(x > (x_max/steps)*(i-1)) & (x <= (x_max/steps)*i)]
97 X_CandidatePeak = X[np.argmax(Y)]
98 Y_CandidatePeak = np.max(Y)
99 Y1_CandidatePeak = Y1[np.argmax(Y)]
100 Y2_CandidatePeak = Y2[np.argmax(Y)]
101
102 a = 1.1*np.max(Y)/2           # Threshold on Background
103 b = 0.9*np.max(abs(Y1))/2    # Threshold on Slope
104 c = 0.1*np.max(abs(Y2))/2    # Threshold on Concavity
105
106 if (Y_CandidatePeak > a)and(abs(Y1_CandidatePeak)< b)and(Y2_CandidatePeak<-c)and(
107     Y_CandidatePeak > 0.13*np.max(y)):
108
109     for j in range(len(X_Peaks1)):
110         if X_CandidatePeak == X_Peaks1[j]:
111             X_Peaks2 = np.append(X_Peaks2, X_CandidatePeak)
112             Y_Peaks2 = np.append(Y_Peaks2, Y_CandidatePeak)
113
114 # Showing determined peaks
115 plt.figure('Raw'+SourceName+' with determined peaks')
116 plt.plot(x,y,'ko',markersize=0.5)

```

```

115 plt.xlim((1, x_max))
116 plt.xlabel('Channels')
117 bottom, top = plt.ylim()
118 plt.ylabel('Counts')
119 plt.title('RAW Data with determined peaks ['+SourceName+']')
120 plt.ticklabel_format(axis='both', style='sci', scilimits=(0,0), useMathText=True)
121 plt.show()
122 plt.plot(X_Peaks2, Y_Peaks2, 'X', c='orange', markeredgecolor='k', markersize=10, alpha=0.75)
123
124 ## Principal-Peak [P.P] Analysis
125
126 xmax_peak = X_Peaks2[np.argmax(Y_Peaks2)] # Principal-Peak position
127 ymax_peak = np.max(Y_Peaks2) # Principal-Peak value
128 index = np.where(x==xmax_peak)[0] # Finding P.P in our data array
129
130 ##### Definition of a ROI: we must have |yROI-yPeak|<=d and |yROI'|<t, on both sides of P.P.
131 ##### Best Values [HEURISTIC]
132 d = ymax_peak*0.75 # -----
133 t = 50 # BEST VALUES [HEURISTIC]
134 #####
135
136
137 x1=x[(abs(y1)<t) & (x<xmax_peak) & (abs(y-ymax_peak)>d)]
138 index1=np.where(x==x1[abs(x1-xmax_peak)==min(abs(x1-xmax_peak))])[0]
139 i1=np.array(index1[0])
140 x2=x[(abs(y1)<t) & (x>xmax_peak) & (abs(y-ymax_peak)>d)]
141 index2=np.where(x==x2[abs(x2-xmax_peak)==min(abs(x2-xmax_peak))])[0]
142 i2=np.array(index2[0])
143 xnew=x[i1:i2]
144 ynew=y[i1:i2]
145
146 #Plotting the ROI-confined Peak
147 plt.figure('Peak-Fitting_'+SourceName)
148 plt.plot(x[range(i1-1,i2+1)],y[range(i1-1,i2+1)],'ko',markersize=1.5,label='ROI-data')
149
150 # Background Esimate: Linear-Fit beetwen ROI edges...
151 a=xnew[0]
152 b=xnew[len(xnew)-1]
153 x_to_fit = np.array([xnew[0],xnew[len(xnew)-1]])
154 y_to_fit = np.array([ynew[0],ynew[len(xnew)-1]])
155 coefficients = np.polyfit(x_to_fit, y_to_fit, deg=1) # Linear-Fit
156 polynomial = np.poly1d(coefficients) # -----
157 x_pol = np.linspace(xnew[0],xnew[len(xnew)-1],100)
158 y_pol = polynomial(x_pol)
159 plt.plot(x_pol,y_pol,'m-.', linewidth=1.5, alpha=0.5, label='Background')
160
161 # Background-correcting the Peak [and plotting]
162 ynew=ynew-polynomial(xnew)
163 plt.plot(xnew,ynew,'o',c='orange',markersize=1.5,label='Corrected-Peak')
164 plt.ylim(0,ymax_peak*2-1.25*d)
165
166 ## Gaussian Best-Fit of the corrected P.P. data
167
168 def Gauss(x,A,x0,sigma):
169     return A*np.exp(-(x-x0)**2/(2*sigma**2))
170
171 A_init = np.max(ynew) # -----
172 x0_init = np.mean(xnew) # First guess
173 sigma_init = np.std(xnew) # -----

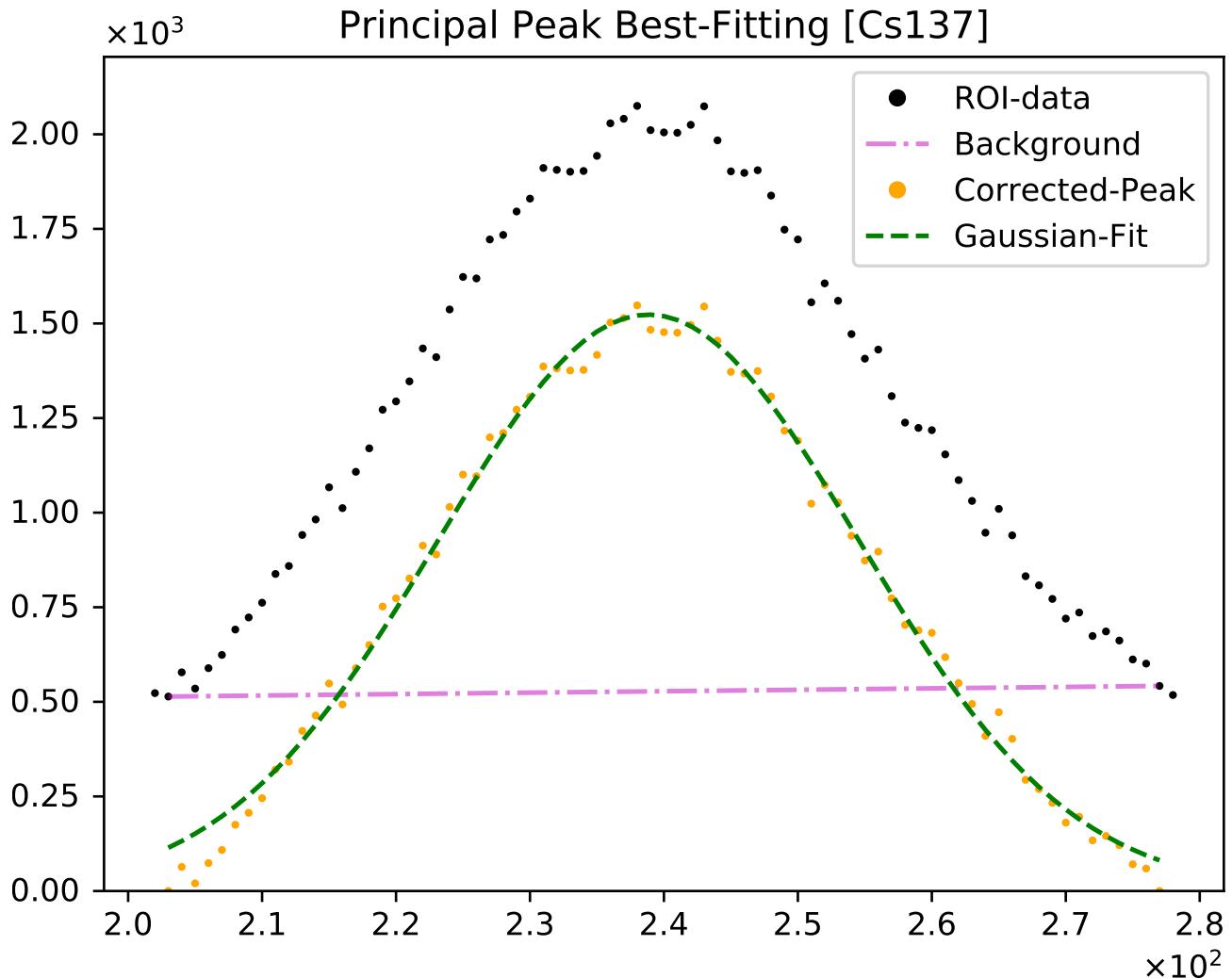
```

```

174
175 # Least-Squares Fitting...
176 popt,pcov = curve_fit(Gauss,xnew,ynew,p0=[A_init,x0_init,sigma_init])
177
178 A = popt[0] # -----
179 x0 = popt[1] # Best-Fit results
180 sigma = popt[2] # -----
181 StDev_vec = 2.3548200*np.sqrt(np.diag(pcov)) # -----
182
183 W = 2.3548200*popt[2] # Peak-Width: FWHM
184 S = 0.5*A*W # Area with Triangle-Approx.
185 R = W/x0 # Resolution: FWHM/x0
186
187 dA = StDev_vec[0]
188 dx = StDev_vec[1]
189 dW = 2.3548200*StDev_vec[2]
190 dS = (dA/A + dW/W)*S
191 dR = (dW/W + dx/x0)*R
192
193 # Printing Results...
194 print('PRINCIPAL-PEAK POSITION', x0,'''$\pm$'', dx)
195 print('PRINCIPAL-PEAK HEIGHT: ', A,'''$\pm$'', dA)
196 print('PRINCIPAL-PEAK WIDTH: ', W,'''$\pm$'', dW)
197 print('PRINCIPAL-PEAK AREA :', S,'''$\pm$'', dS)
198 print('CRYSTAL RESOLUTION: ', R,'''$\pm$'', dR)
199
200 # Plotting & Showing All
201 yFIT = Gauss(xnew,A,x0,sigma)
202 plt.plot(xnew,yFIT,'g--', linewidth=1.5, label='Gaussian-Fit')
203 plt.title('Principal Peak Best-Fitting ['+SourceName+']')
204 plt.ticklabel_format(axis='both', style='sci', scilimits=(0,0), useMathText=True)
205 plt.legend(markerscale=3)
206 plt.show()

```

Figura 21: Analisi su dati grezzi del Picco Principale (P.P.) della sorgente Cs¹³⁷.

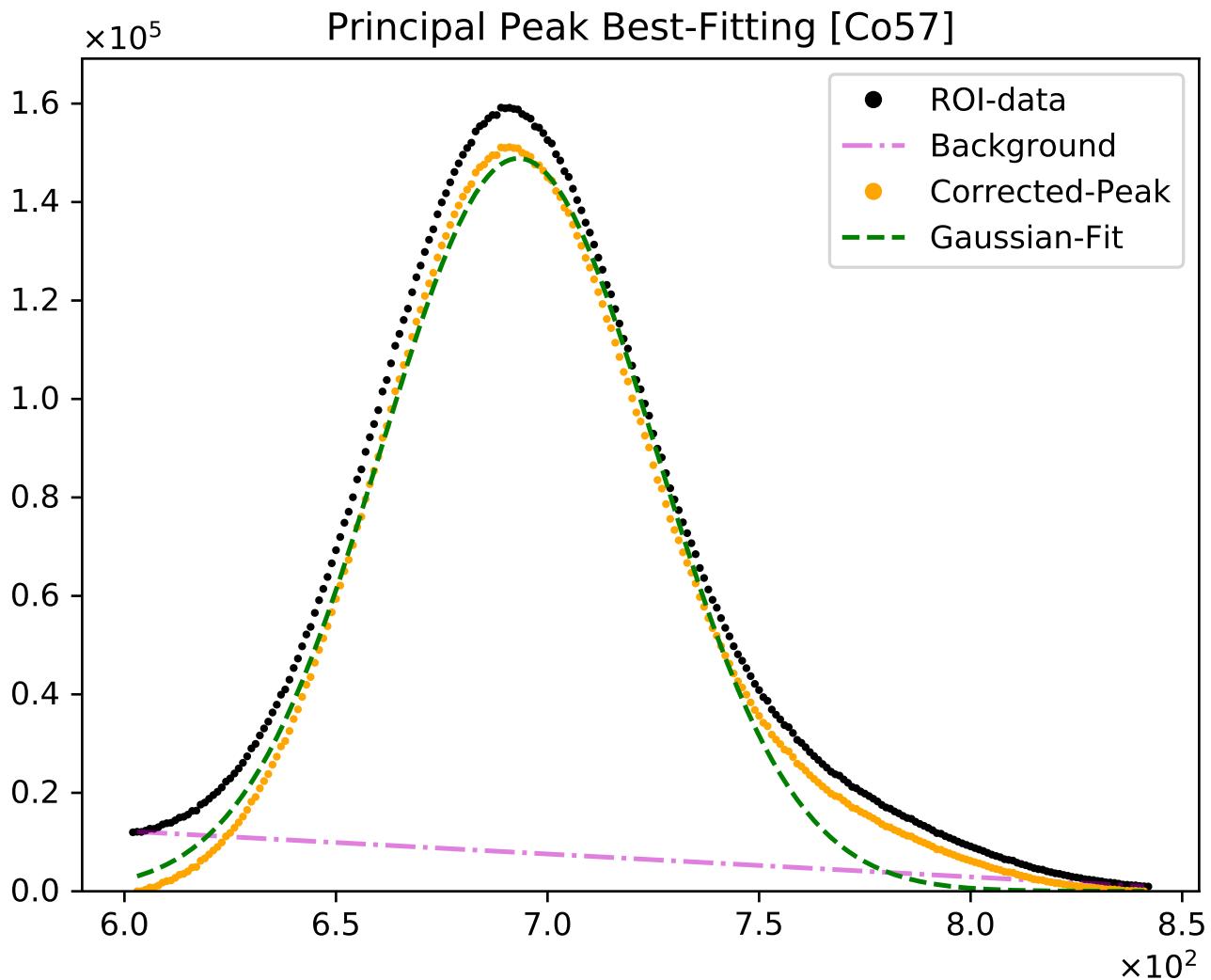


```

1  >>> (executing file "PhytonCodeEs5.py")
2  Insert the desired source [Cs137|Co57|Na22|Ba133|Eu152]: Cs137
3  PRINCIPAL-PEAK POSITION 238.8467517967467 ± 0.3098839895739907
4  PRINCIPAL-PEAK HEIGHT: 1523.3400945401845 ± 26.00320803587605
5  PRINCIPAL-PEAK WIDTH: 37.09865495020793 ± 0.7485587193418013
6  PRINCIPAL-PEAK AREA : 28256.934269581718 ± 1052.4967764062292
7  CRYSTAL RESOLUTION: 0.15532409241963677 ± 0.003335574642671817
8

```

Figura 22: Analisi su dati grezzi del Picco Principale (P.P.) della sorgente Co⁵⁷.

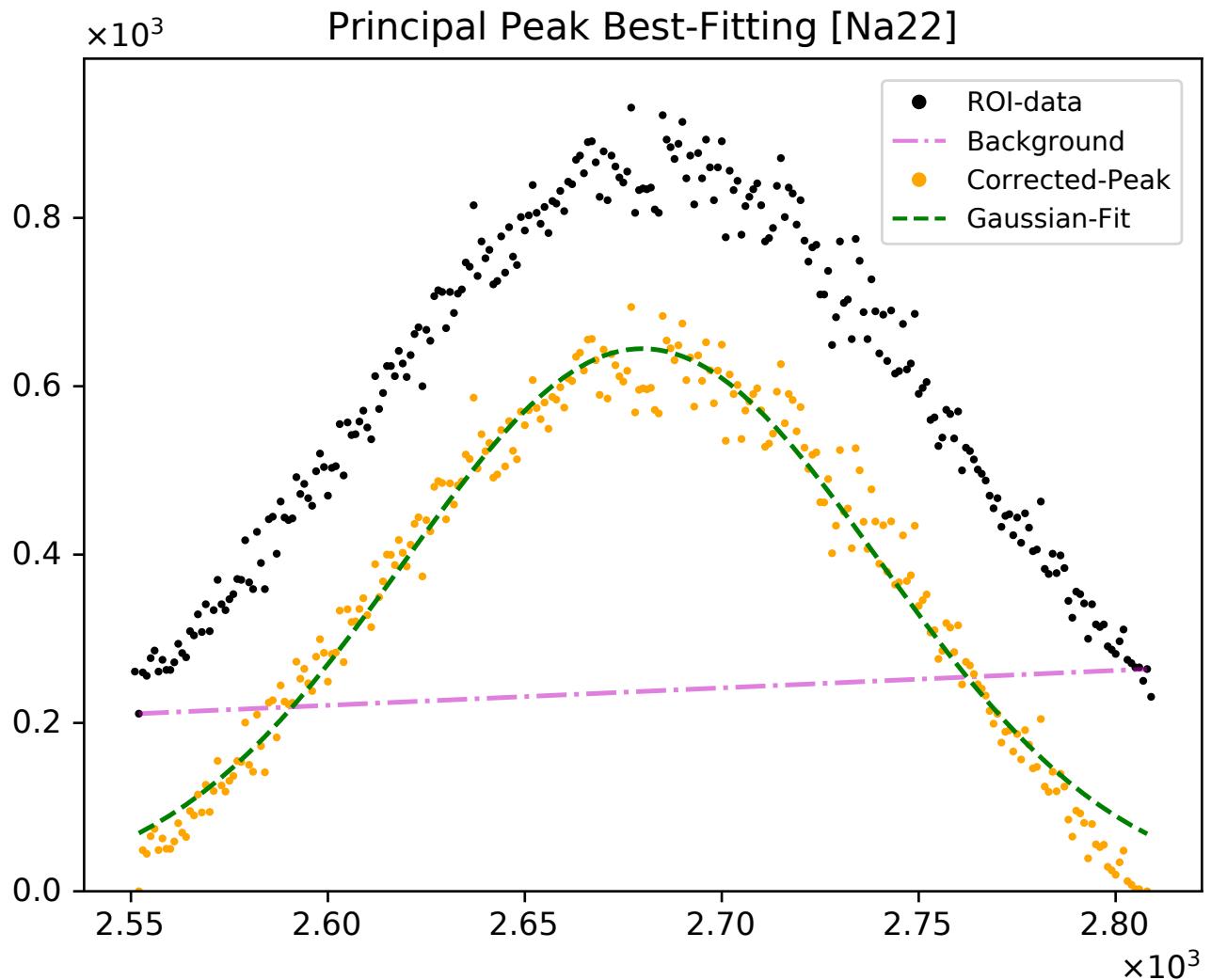


```

1  >>> (executing file "PhytonCodeEs5.py")
2  Insert the desired source [Cs137|Co57|Na22|Ba133|Eu152]: Co57
3  PRINCIPAL-PEAK POSITION 693.1137861569199 ± 0.4433360809406253
4  PRINCIPAL-PEAK HEIGHT: 148880.4511121081 ± 1769.474256713379
5  PRINCIPAL-PEAK WIDTH: 76.09652174560256 ± 1.04632416144232
6  PRINCIPAL-PEAK AREA : 5664642.242773826 ± 145214.0247096522
7  CRYSTAL RESOLUTION: 0.10978936397663056 ± 0.0015798239331929512
8

```

Figura 23: Analisi su dati grezzi del Picco Principale (P.P.) della sorgente Na^{22} .

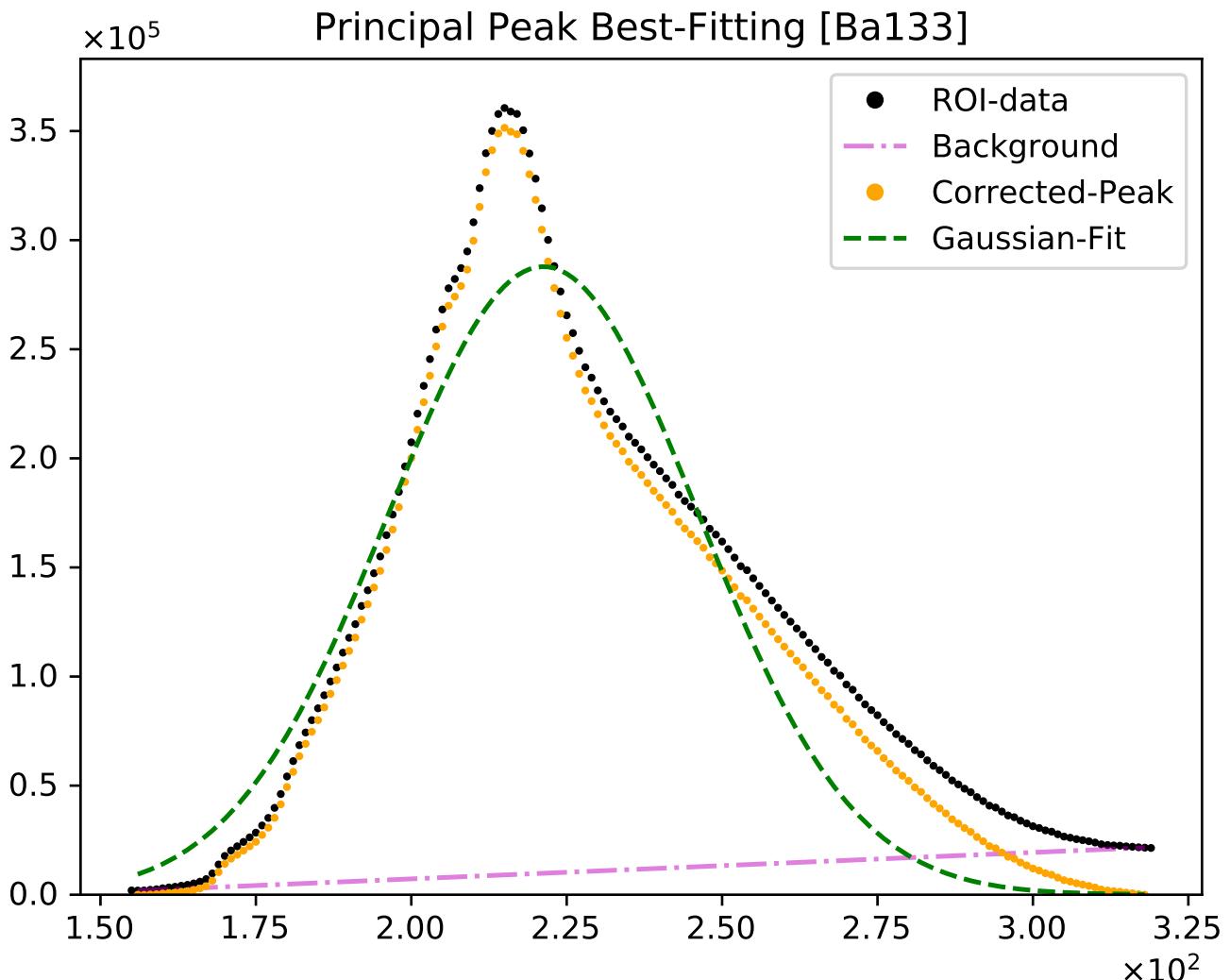


```

1  >>> (executing file "PhytonCodeEs5.py")
2  Insert the desired source [Cs137|Co57|Na22|Ba133|Eu152]: Na22
3  PRINCIPAL-PEAK POSITION 2679.8233949851183 ± 1.0636289671764676
4  PRINCIPAL-PEAK HEIGHT: 644.6055128147791 ± 9.837826754288718
5  PRINCIPAL-PEAK WIDTH: 142.39273009203848 ± 2.6530917116409816
6  PRINCIPAL-PEAK AREA : 45893.56940103744 ± 1555.5162765213215
7  CRYSTAL RESOLUTION: 0.05313511717171542 ± 0.0010111143019759258
8

```

Figura 24: Analisi su dati grezzi del Picco Principale (P.P.) della sorgente Ba¹³³.

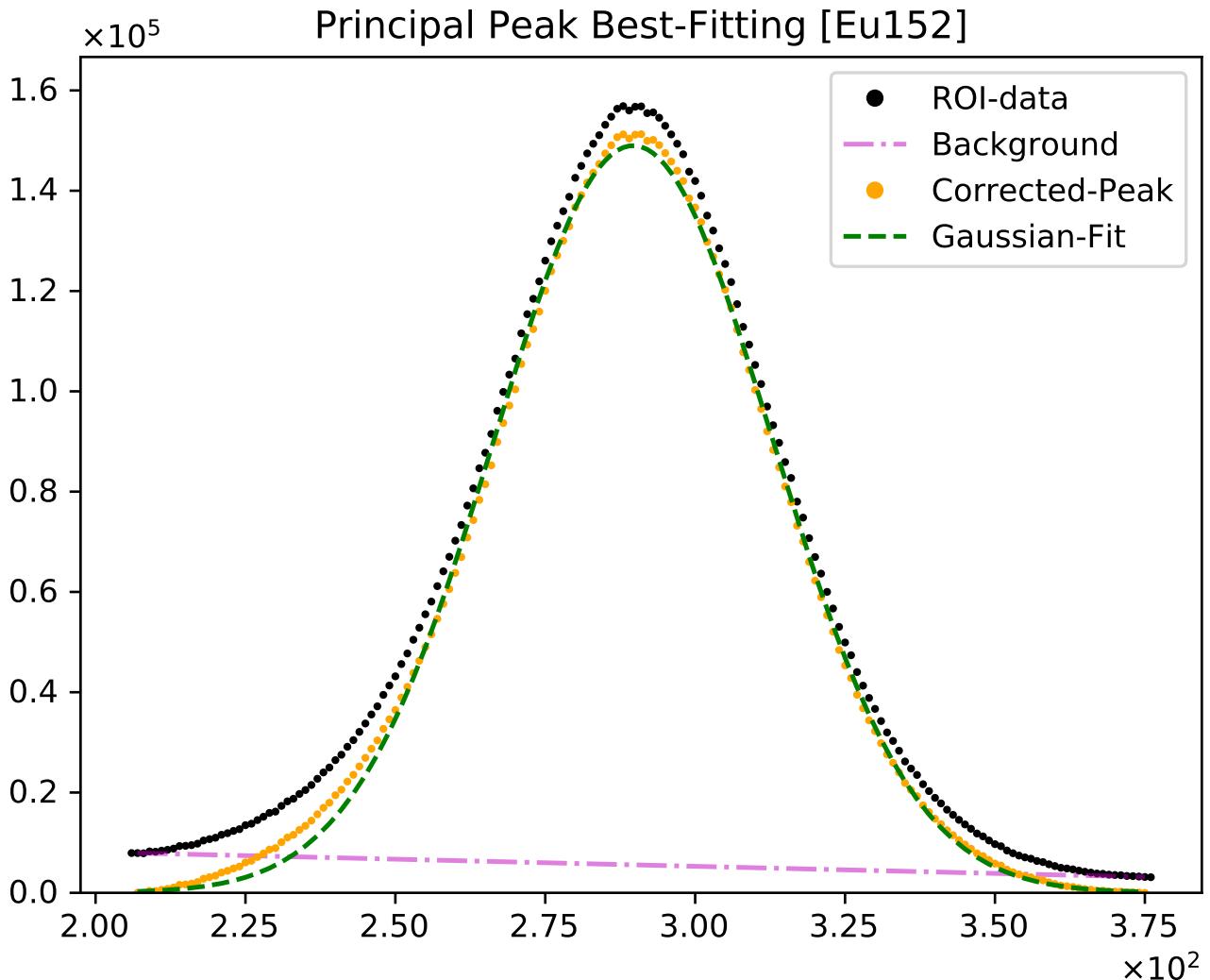


```

1  >>> (executing file "PhytonCodeEs5.py")
2  Insert the desired source [Cs137|Co57|Na22|Ba133|Eu152]: Ba133
3  PRINCIPAL-PEAK POSITION 221.25212247553907 ± 1.2732933530258672
4  PRINCIPAL-PEAK HEIGHT: 287864.93020777084 ± 12741.050893987342
5  PRINCIPAL-PEAK WIDTH: 58.70101313148694 ± 3.0122036998891812
6  PRINCIPAL-PEAK AREA : 8448981.524110464 ± 807510.2018385414
7  CRYSTAL RESOLUTION: 0.2653127684141278 ± 0.01514120925440681
8

```

Figura 25: Analisi su dati grezzi del Picco Principale (P.P.) della sorgente Eu¹⁵².



```

1  >>> (executing file "PhytonCodeEs5.py")
2  Insert the desired source [Cs137|Co57|Na22|Ba133|Eu152]: Eu152
3  PRINCIPAL-PEAK POSITION 289.67056608195855 ± 0.15968983090396094
4  PRINCIPAL-PEAK HEIGHT: 149000.1993161386 ± 887.0254434869172
5  PRINCIPAL-PEAK WIDTH: 54.704433482585735 ± 0.3760595236985809
6  PRINCIPAL-PEAK AREA : 4075485.7461908604 ± 52278.58417820594
7  CRYSTAL RESOLUTION: 0.18885050774232898 ± 0.0014023414074840683
8

```

Complessivamente l'analisi operata sui dati grezzi risulta ragionevole: i valori di incertezza sui parametri di *best-fit* sono almeno di due ordini di grandezza inferiori al relativo valore stimato in tutti casi meno che per il P.P. del Ba¹³³, che ha profilo palesemente non gaussiano. Osserviamo che la risoluzione spettrale dei picchi risulta migliore per le risonanze a energie inferiori, per cui deduciamo che il cristallo abbia risoluzione decrescente con l'energia di eccitazione: di ciò abbiamo trovato riscontro in letteratura²⁴. Per quanto riguarda le proprietà di risposta del cristallo abbiamo invece confrontato direttamente gli istogrammi grezzi in Figura 19 con gli spettri di emissione riportati su [Gamma Spectacular](#) per alcune delle sorgenti considerate²⁵ e dedotto che il cristallo non presenta una buona linearità se non nel limite di alte energie.

* * *

Per operare lo smoothing dei dati si è adoperata la libreria `scipy.ndimage.gaussian_filter`²⁶ di Python, ottenendo i risultati riportati in Figura 26 secondo il codice di seguito riportato:

```

1  """ Gaussian-Smoothing Library """
2  from scipy.ndimage import gaussian_filter as smoothing
3
4  ## Smoothing Data
5  y=smoothing(data,sigma=5)
6
7  ## Plotting Results
8  plt.figure('Smoothing'+SourceName)
9  plt.plot(x,data,'ko',markersize=0.5,label='RAW Data')
10 plt.plot(x,y,'ro',markersize=0.5,label='Smoothed')
11 plt.xlim((1, x_max))
12 plt.xlabel('Channels')
13 plt.ylim((bottom, top))
14 plt.ylabel('Counts')
15 plt.title('Smoothing ['+SourceName+']')
16 plt.legend(markerscale=10)
17 plt.ticklabel_format(axis='both',style='sci',scilimits=(0,0),useMathText=True)
18 plt.figure('Smoothed'+SourceName)
19 plt.plot(x,y,'ro',markersize=0.5)
20 plt.xlim((1, x_max))
21 plt.xlabel('Channels')
22 plt.ylim((bottom, top))
23 plt.ylabel('Counts')
24 plt.title('Smoothed Data with determined peaks ['+SourceName+']')
25 plt.ticklabel_format(axis='both',style='sci',scilimits=(0,0),useMathText=True)
26 plt.show()

```

Listing 13: Python code for Gaussian Smoothing Routine

L'applicazione della medesima routine (Listato 12 a partire dalla riga 50) sui dati sottoposti a smoothing porta a individuare i picchi riportati in Figura 27, in cui possiamo osservare come le risonanze trascurate in precedenza su Co⁵⁷ e Na²² vengano adesso correttamente riconosciute. Infine osserviamo che l'analisi del P.P. risulta notevolmente migliorata dal processo di smoothing, con un generale miglioramento delle incertezze associate ai parametri del modello Gaussiano. Tuttavia possiamo notare come l'altezza dei picchi si riduca sistematicamente passando dai dati grezzi a quelli sottoposti a smoothing e viceversa cresca la larghezza dei picchi: ne deduciamo che lo smoothing, pur aiutando nella determinazione e analisi dei picchi, introduce dei bias sulla stima di alcuni parametri fisici, quali ad esempio la risoluzione spettrale del cristallo.

²⁴G. Mishra *et al.*, Proc. DAE Symp. Nucl. Phys. 62 (2017) 1092-1093

F. Quarati *et al.*, Nuclear Instruments and Methods in Physics Research A 729 (2013) 596–604

²⁵In particolare per Na²² e Ba¹³³ riconoscendone i picchi rispettivamente a 511 keV e 1274 keV e a 31 keV e 81 keV.

²⁶Documentazione: https://docs.scipy.org/doc/scipy/reference/generated/scipy.ndimage.gaussian_filter.html

Figura 26: Smoothing dei cinque spettri di calibrazione considerati.

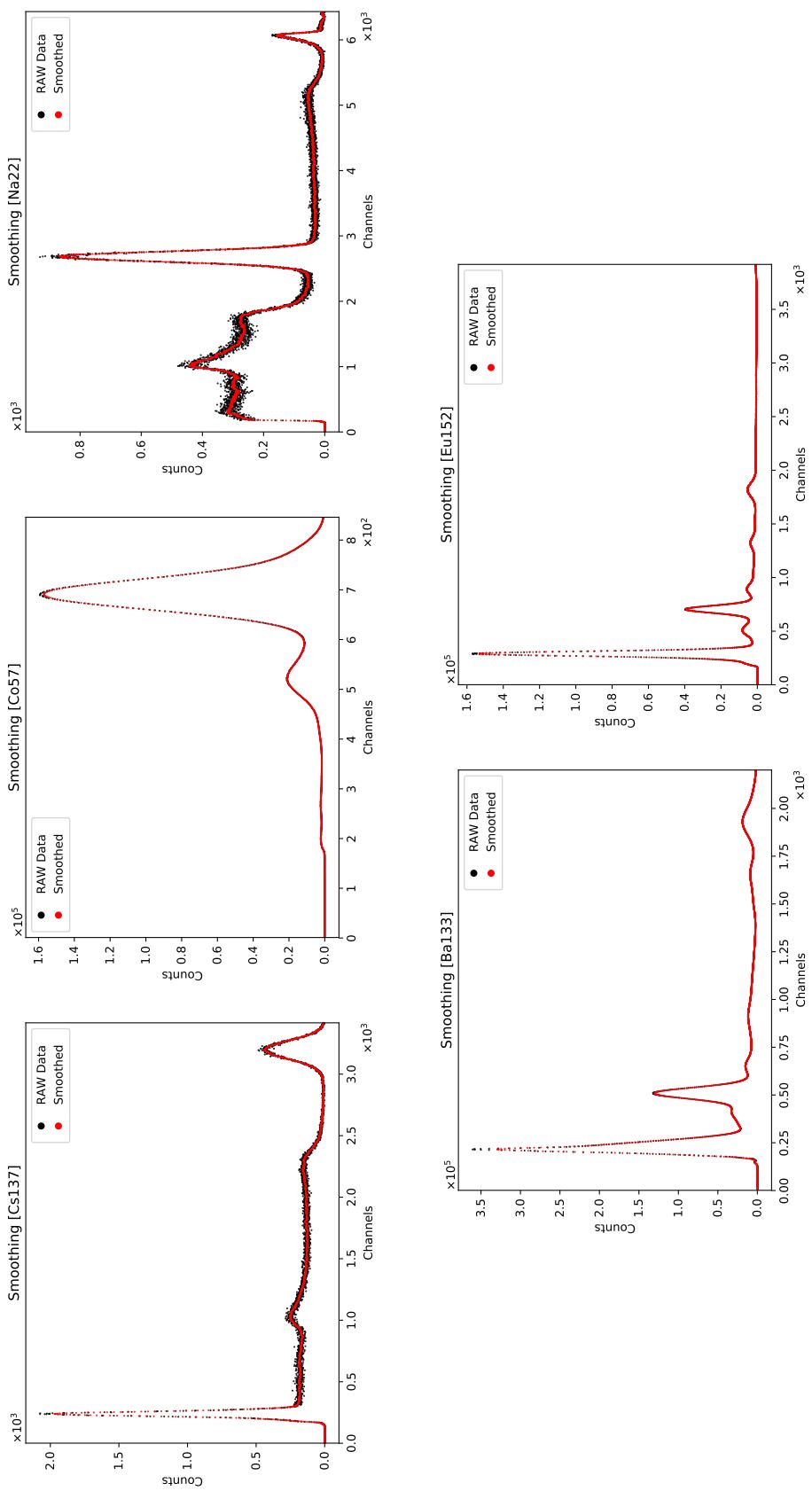


Figura 27: Ricerca dei picchi di scintillazione sugli spettri di calibrazione sottoposti a smoothing.

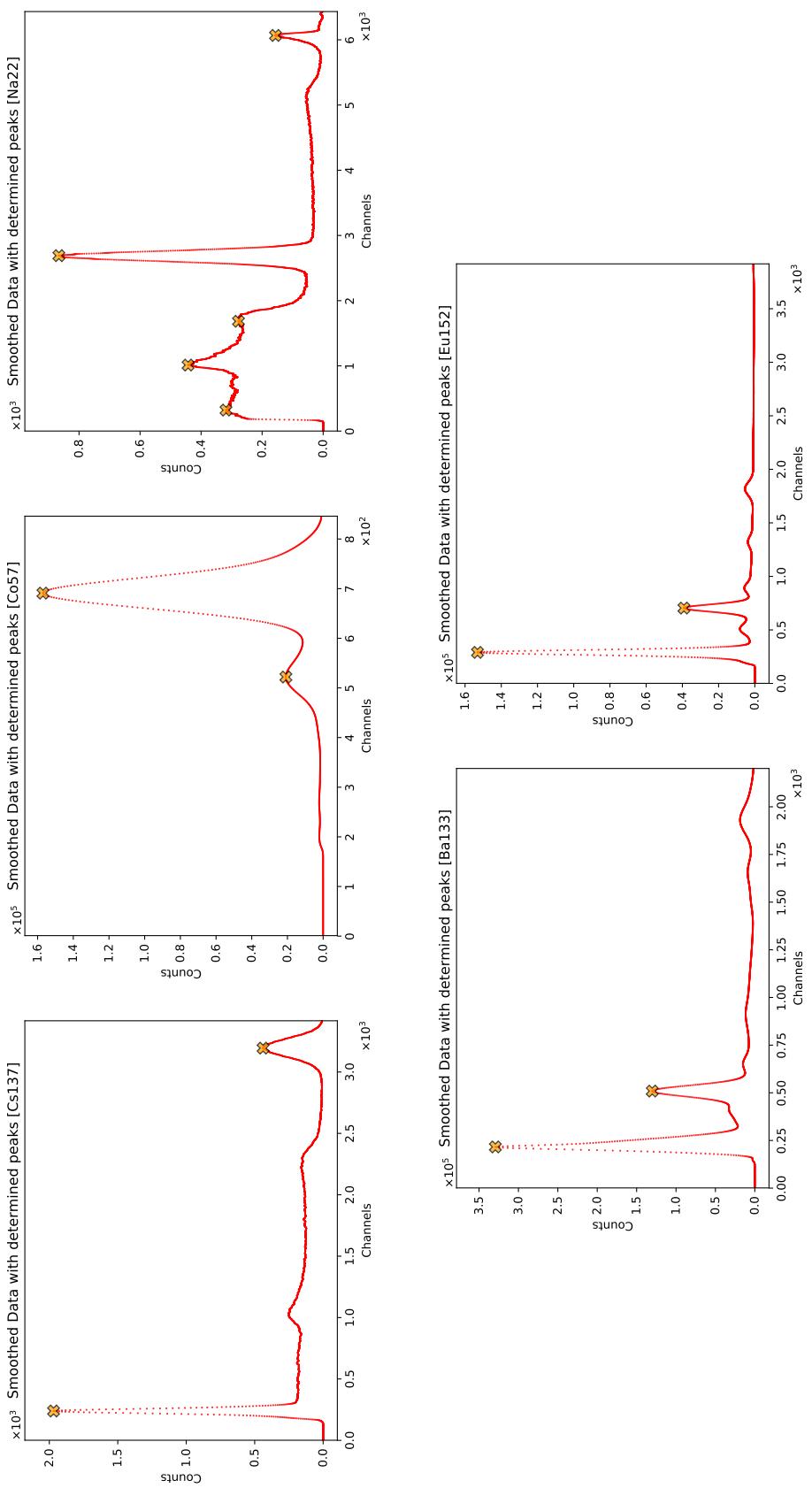
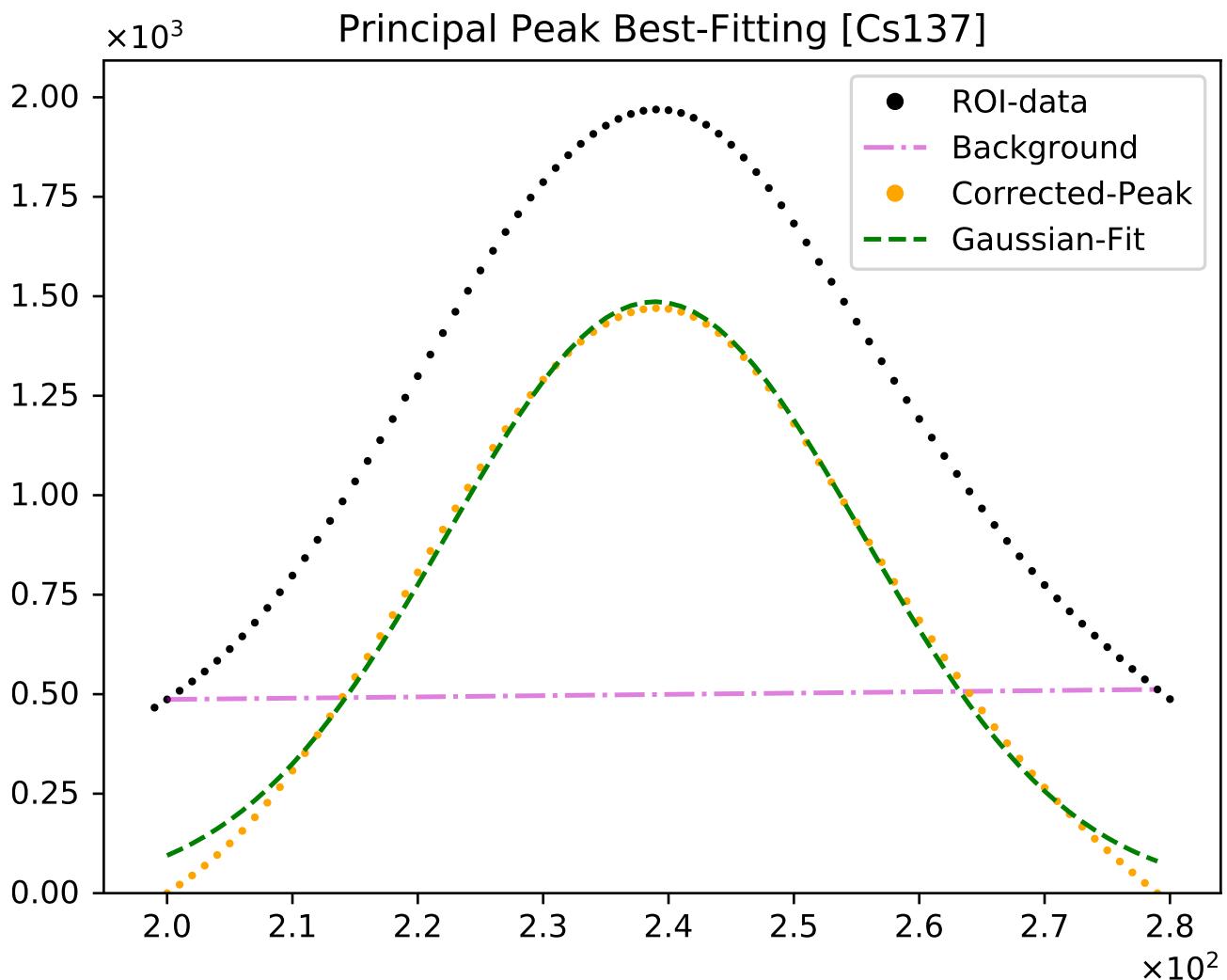


Figura 28: Analisi su dati *smoothed* del Picco Principale (P.P.) della sorgente Cs¹³⁷.

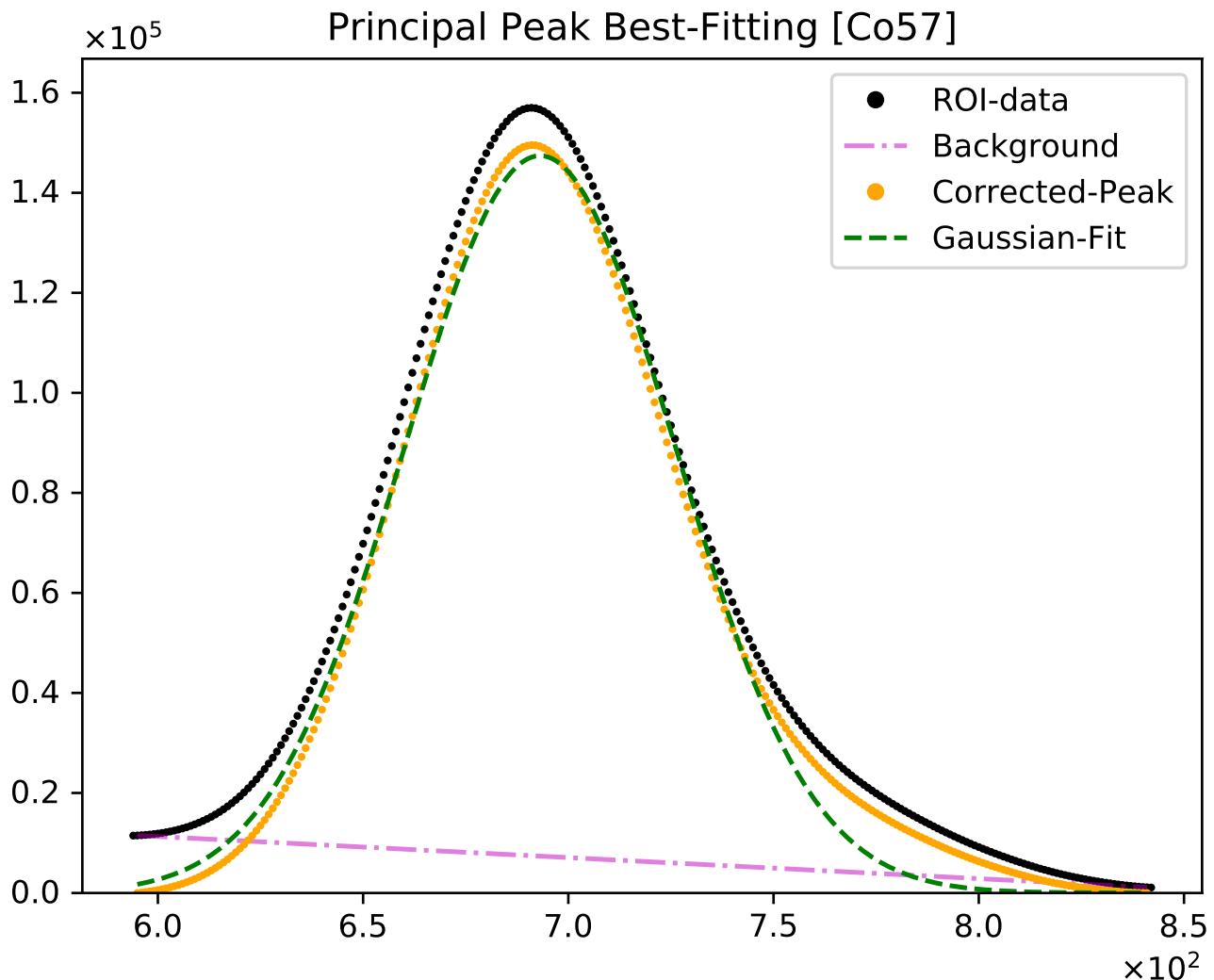


```

1  >>> (executing file "PhytonCodeEs5.py")
2  Insert the desired source [Cs137|Co57|Na22|Ba133|Eu152]: Cs137
3  PRINCIPAL-PEAK POSITION 238.91761912451213 ± 0.22132312943379565
4  PRINCIPAL-PEAK HEIGHT: 1486.306984085268 ± 17.214673725913784
5  PRINCIPAL-PEAK WIDTH: 39.04930494268223 ± 0.5329330173241219
6  PRINCIPAL-PEAK AREA : 29019.62732999198 ± 732.1615547552321
7  CRYSTAL RESOLUTION: 0.16344255013830372 ± 0.0023820203637086343
8

```

Figura 29: Analisi su dati *smoothed* del Picco Principale (P.P.) della sorgente Co⁵⁷.

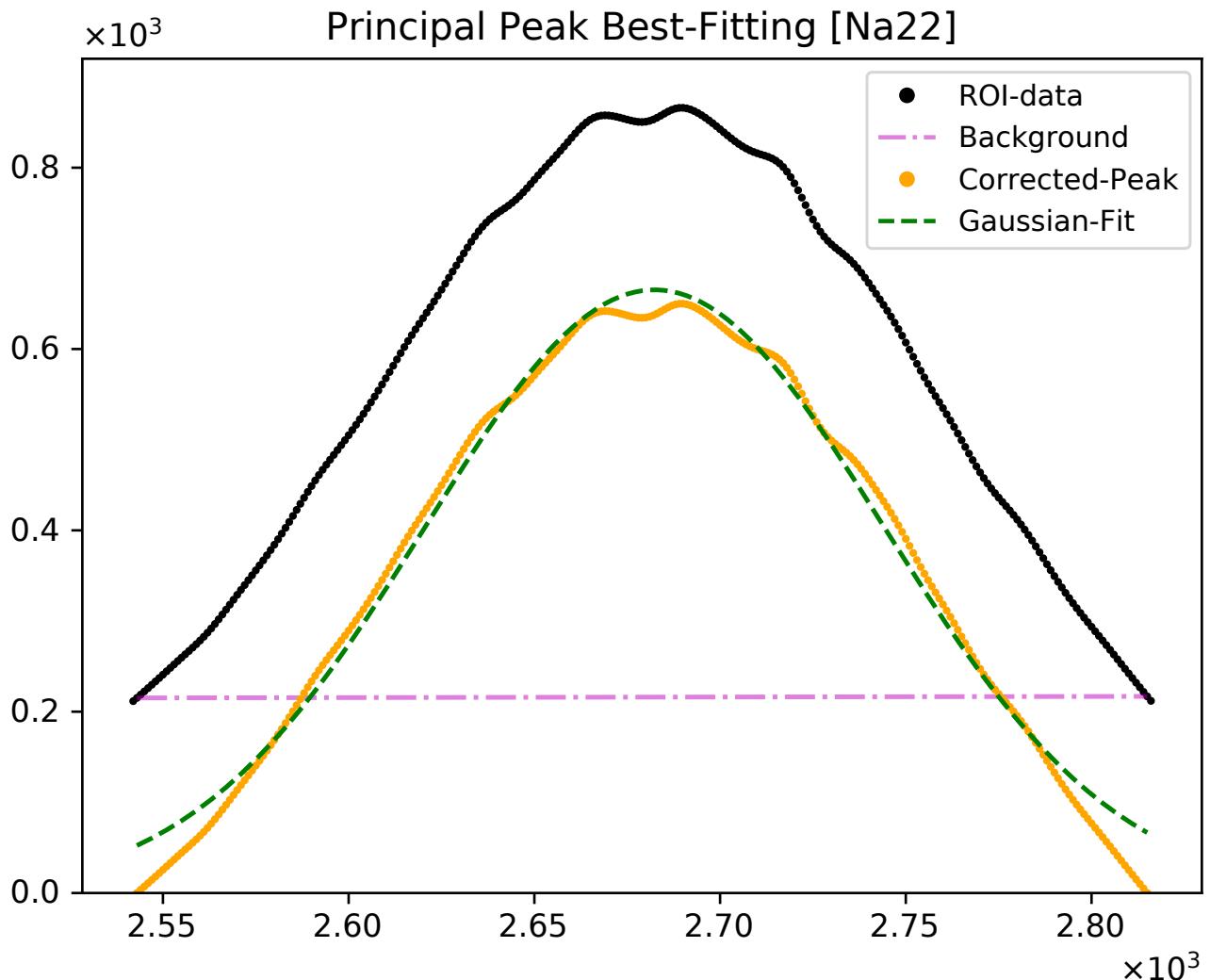


```

1  >>> (executing file "PhytonCodeEs5.py")
2  Insert the desired source [Cs137|Co57|Na22|Ba133|Eu152]: Co57
3  PRINCIPAL-PEAK POSITION 693.1196439613678 ± 0.42736955821149747
4  PRINCIPAL-PEAK HEIGHT: 147417.44636429526 ± 1658.6905056705684
5  PRINCIPAL-PEAK WIDTH: 77.4723612953905 ± 1.007283111318557
6  PRINCIPAL-PEAK AREA : 5710388.832989266 ± 138496.887084504
7  CRYSTAL RESOLUTION: 0.11177343186035649 ± 0.0015221782887043748
8

```

Figura 30: Analisi su dati *smoothed* del Picco Principale (P.P.) della sorgente Na^{22} .

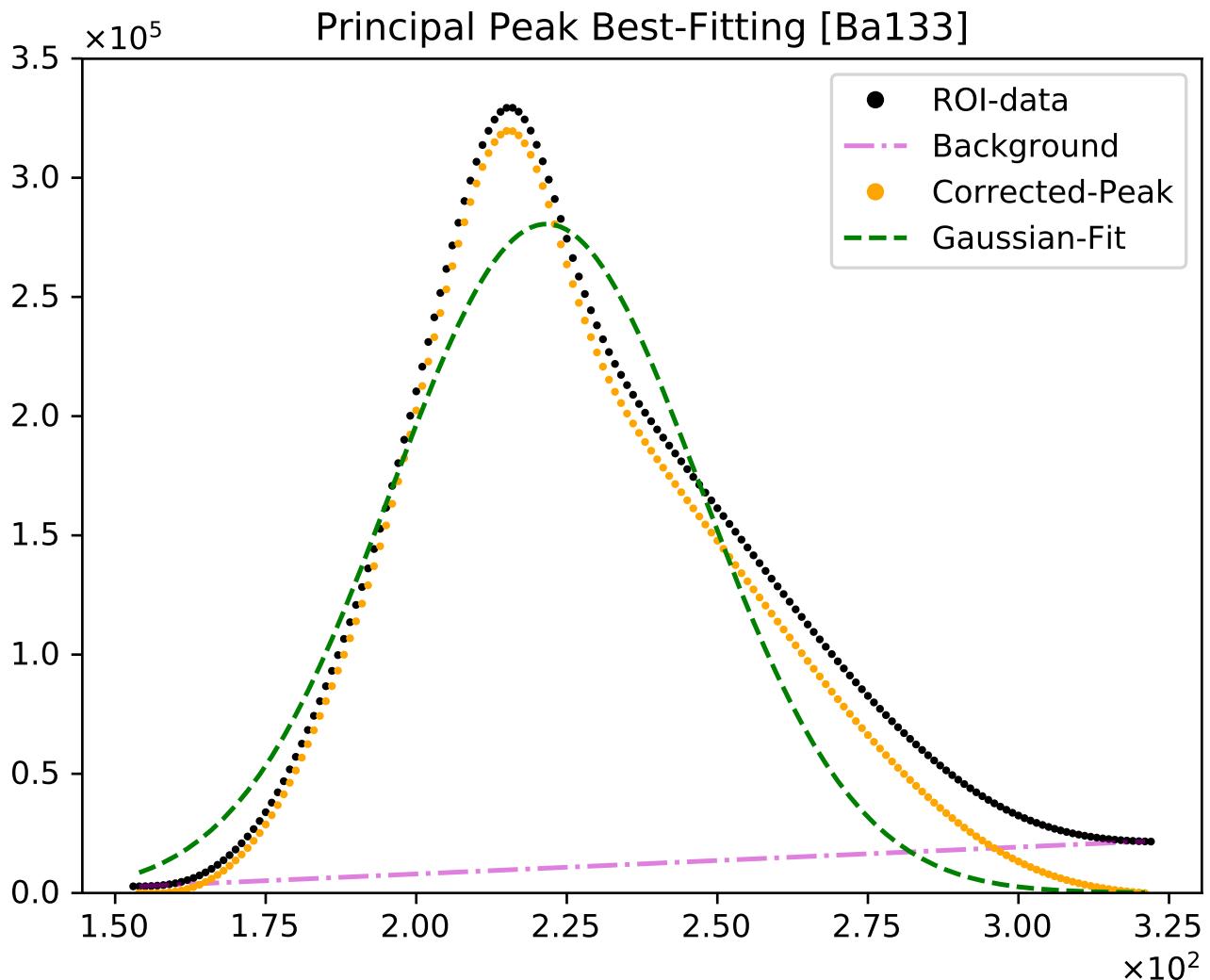


```

1  >>> (executing file "PhytonCodeEs5.py")
2  Insert the desired source [Cs137|Co57|Na22|Ba133|Eu152]: Na22
3  PRINCIPAL-PEAK POSITION 2682.367779401036 ± 0.6482369680134931
4  PRINCIPAL-PEAK HEIGHT: 665.286115220424 ± 6.054778595618799
5  PRINCIPAL-PEAK WIDTH: 145.5665828324627 ± 1.596600338802527
6  PRINCIPAL-PEAK AREA : 48421.71319926059 ± 971.7847334664561
7  CRYSTAL RESOLUTION: 0.054267943400724585 ± 0.0006083352321870312
8

```

Figura 31: Analisi su dati *smoothed* del Picco Principale (P.P.) della sorgente Ba¹³³.

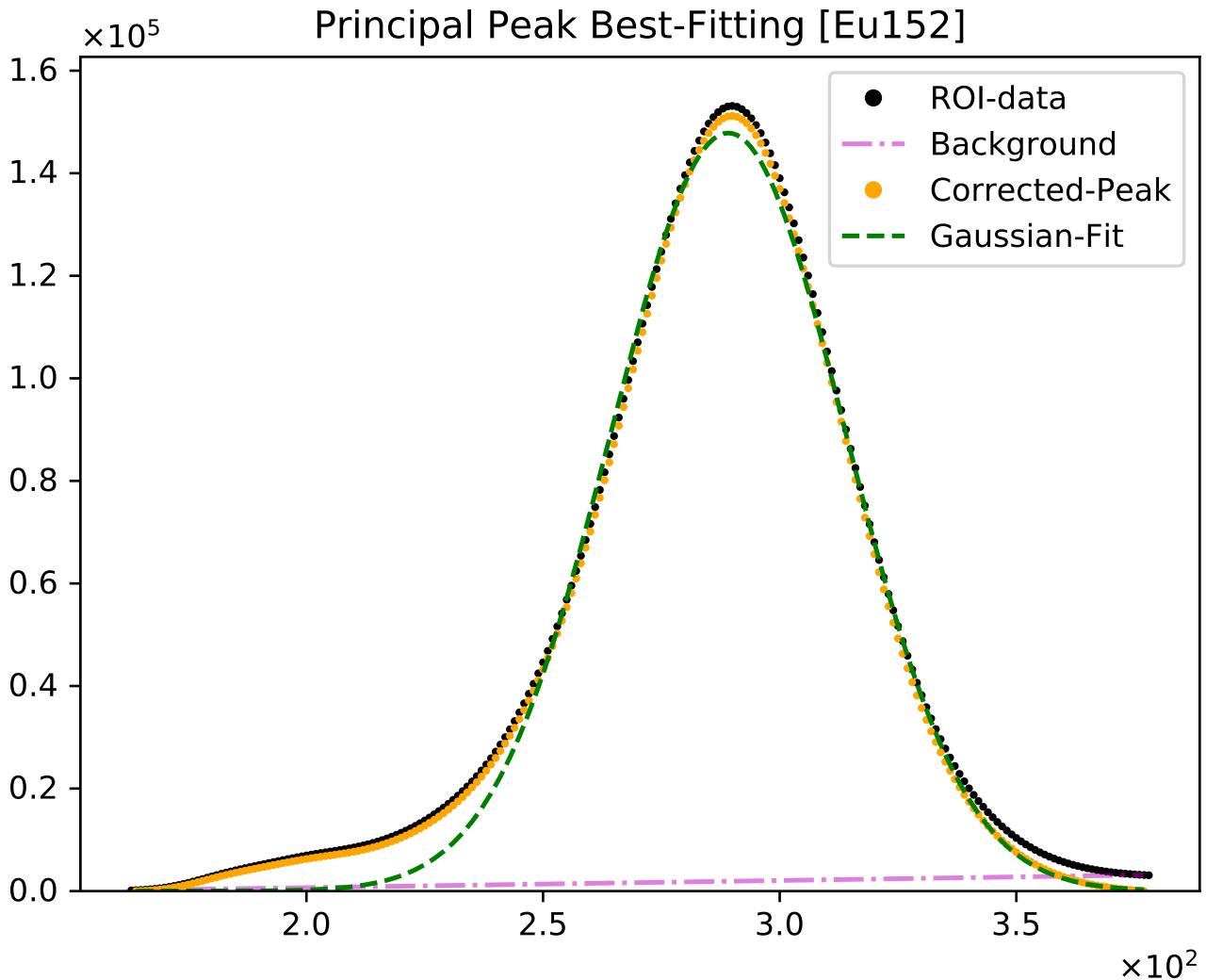


```

1  >>> (executing file "PhytonCodeEs5.py")
2  Insert the desired source [Cs137|Co57|Na22|Ba133|Eu152]: Ba133
3  PRINCIPAL-PEAK POSITION 221.61534504437415 ± 1.1266699889196987
4  PRINCIPAL-PEAK HEIGHT: 280582.0153386667 ± 10704.239741249454
5  PRINCIPAL-PEAK WIDTH: 60.25857423710266 ± 2.6641739924518424
6  PRINCIPAL-PEAK AREA : 8453736.100440461 ± 696270.7665574122
7  CRYSTAL RESOLUTION: 0.27190614542073815 ± 0.013403956687680785
8

```

Figura 32: Analisi su dati *smoothed* del Picco Principale (P.P.) della sorgente Eu¹⁵².



```

1  >>> (executing file "PhytonCodeEs5.py")
2  Insert the desired source [Cs137|Co57|Na22|Ba133|Eu152]: Eu152
3  PRINCIPAL-PEAK POSITION 289.12654333546305 ± 0.32331257942848335
4  PRINCIPAL-PEAK HEIGHT: 147846.4568358735 ± 1672.005453919737
5  PRINCIPAL-PEAK WIDTH: 58.302795781850186 ± 0.7613703593365995
6  PRINCIPAL-PEAK AREA : 4309930.889986031 ± 105024.2512468971
7  CRYSTAL RESOLUTION: 0.20165148142141887 ± 0.002858841012675539
8

```

Esercizio 6

Si consideri un reticolo di diffrazione con $N = 5$ fenditure. Si supponga di avere uno schermo a distanza L dalla sorgente. Si simuli con un programma lo spettro di intensità \mathcal{I} su questo schermo. Per un campione di 50000 eventi si disegni un istogramma sperimentale di \mathcal{I} con *bin-width* Δx scelta opportunamente. A questo punto si applichi uno *smearing gaussiano* con $\sigma = c\Delta x$ ($0 < c < 10$) e si discuta il relativo effetto in funzione della scelta di c . Applicando una delle tecniche di regolarizzazione viste a lezione si faccia l'*unfolding* delle distribuzioni sperimentali così costruite, discutendo la distribuzione ricostruita (si può ricorrere al package `RooUnfold` di `ROOT` o simili alternative).

* * *

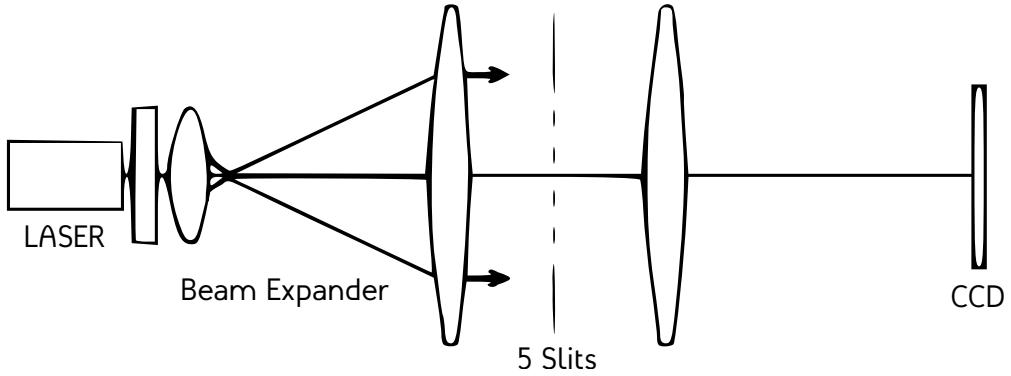
Si è considerato l'esperimento schematizzato in Figura 33, notoriamente ben descritto dalla teoria diffrattiva di campo lontano (diffrazione di Fraunhofer). Tale modello prevede la seguente legge per l'intensità raccolta sullo schermo:

$$\mathcal{I}(\vartheta) = \mathcal{I}_0 \left(\frac{\sin(\beta\vartheta)}{\beta\vartheta} \right)^2 \left(\frac{\sin(N\alpha\vartheta)}{\alpha\vartheta} \right)^2, \quad (8)$$

dove

- ϑ è l'angolo d'incidenza della luce sullo schermo ($\vartheta = 0$ definisce l'asse ottico);
- $\alpha = \frac{ka}{2}$, con a spaziatura fra le N fenditure;
- $\beta = \frac{kb}{2}$, con b larghezza di ogni singola fenditura;
- k è il vettore d'onda della luce monocromatica immessa nel sistema ottico.

Figura 33: Apparato per la rivelazione di profili diffrattivi di campo lontano.



Ponendo in particolare $\mathcal{I}_0 = 1$, $\alpha = 1$, $\beta = 1/2$ e $N = 5$ si ottiene il profilo spettrale tracciato in Figura 34(a).

Per generare un campione distribuito secondo tale legge si è pensato di utilizzare la versione '*hit or miss*' della tecnica Montecarlo: il metodo non è particolarmente efficiente ma di contro risulta estremamente flessibile nel caso di densità di probabilità inusuali.

L'algoritmo per la simulazione della generica densità $g(x)$ può essere sintetizzato nei seguenti passi:

1. Generazione di un numero casuale u , estratto con probabilità uniforme nel dominio di g .
2. Generazione di un numero casuale y , indipendente da u ed estratto uniformemente dall'intervallo $(0, \max[g(x)])$.
3. Criterio '*hit or miss*': si "accetta" u se $y < g(u)$; in caso contrario lo si scarta.
4. Reiterazione *ad libitum* del punto (1).

La famiglia $\{u\}$ così ottenuta sarà distribuita secondo la densità g , dal momento che per ogni valore u generato al passo (1) la probabilità di accettarlo è chiaramente proporzionale a $g(u)$.

In Figura 34(b) è riportato un istogramma degli eventi generati tramite algoritmo '*hit or miss*', con sovrapposta la curva teorica $\mathcal{I}(\vartheta)$. Al netto di un piccolo *bias* rigido (di origine ignota) sulla posizione dei picchi, i dati generati sembrano riprodurre con buona accuratezza lo spettro d'intensità desiderato.

* * *

I dati rappresentati in Figura 34(b) simulano tuttavia solo la luce incidente sul rivelatore, non il segnale effettivamente misurato. In generale quest'ultimo viene in certa misura distorto dalla risposta spettrale del sistema di rivelazione. Chiamati $u(x)$ il segnale "vero" che incide sul rivelatore e $y(x)$ il segnale misurato, nell'assunzione che la risposta in frequenza dello strumento sia lineare possiamo dunque scrivere:

$$y(x_1) = \int_{-\infty}^{+\infty} K(x_1, x_2) u(x_2) dx_2, \quad (9)$$

dove $K(x_1, x_2)$ (detto *kernel*) descrive interamente le caratteristiche di risposta lineare dell'apparato rivelatore.

In generale dunque il kernel associato al rivelatore ne incorpora tutte le informazioni relative a efficienza e limiti di risoluzione²⁷. In particolare una risoluzione finita introduce una probabilità non nulla di registrare erroneamente un eventi relativi al bin j -esimo su un bin i -esimo, coerentemente con l'incertezza associata alla misura. Pertanto, assumendo *accettanza* ideale (tutti gli eventi vengano rivelati in qualche bin) e discretizzando l'equazione (9), otteniamo una relazione matriciale tra frequenze attese sullo schermo $\{\mu\}$ e frequenze effettivamente rivelate $\{\nu\}$:

$$\nu_j = \sum_i [\mathcal{M}_{ij} \mu_i], \quad \mathcal{M}_{ij} = \mathbb{P}\{"\text{Migrazione di un evento dal bin } i\text{-esimo al bin } j\text{-esimo}"\}. \quad (10)$$

La matrice $\{\{\mathcal{M}\}\}$ sarà ovviamente tale da preservare la norma dei vettori su cui è applicata.

Per aggiungere anche gli effetti di risoluzione finita alla nostra simulazione occorre dunque costruire opportunamente una matrice statistica che tenga conto della migrazione aleatoria tra i bin. Per fare ciò introduciamo uno *smearing* gaussiano dei valori binnati, con deviazione standard $\sigma = c\Delta x$, dove Δx indica la larghezza del singolo bin e la costante c controlla il limite risolutivo da simulare.

Lo smearing viene in particolare eseguito generando una sequenza casuale con distribuzione normale, per mezzo del comando `random.normal` incluso nel pacchetto `numpy` di `Python`²⁸. Delle rappresentazioni in *colormap discreta* delle matrici di smearing generate per diversi valori di c sono riportate in Figura 35: i grafici mostrano delle matrici pressocché simmetriche, come ci si aspetta smussando secondo una distribuzione pari rispetto al valor medio.

²⁷Eventuali fondi introdotti durante la rivelazione non vengono descritti dal modello, trattandosi di effetti di risposta non lineare.

²⁸La documentazione relativa può essere consultata al seguente link: <https://docs.scipy.org/doc/.../numpy.random.normal.html>.

Figura 34: (a) Grafico dello spettro di intensità di Fraunhofer in funzione dell'angolo.
 (b) Confronto con l'istogramma degli eventi Montecarlo generati.

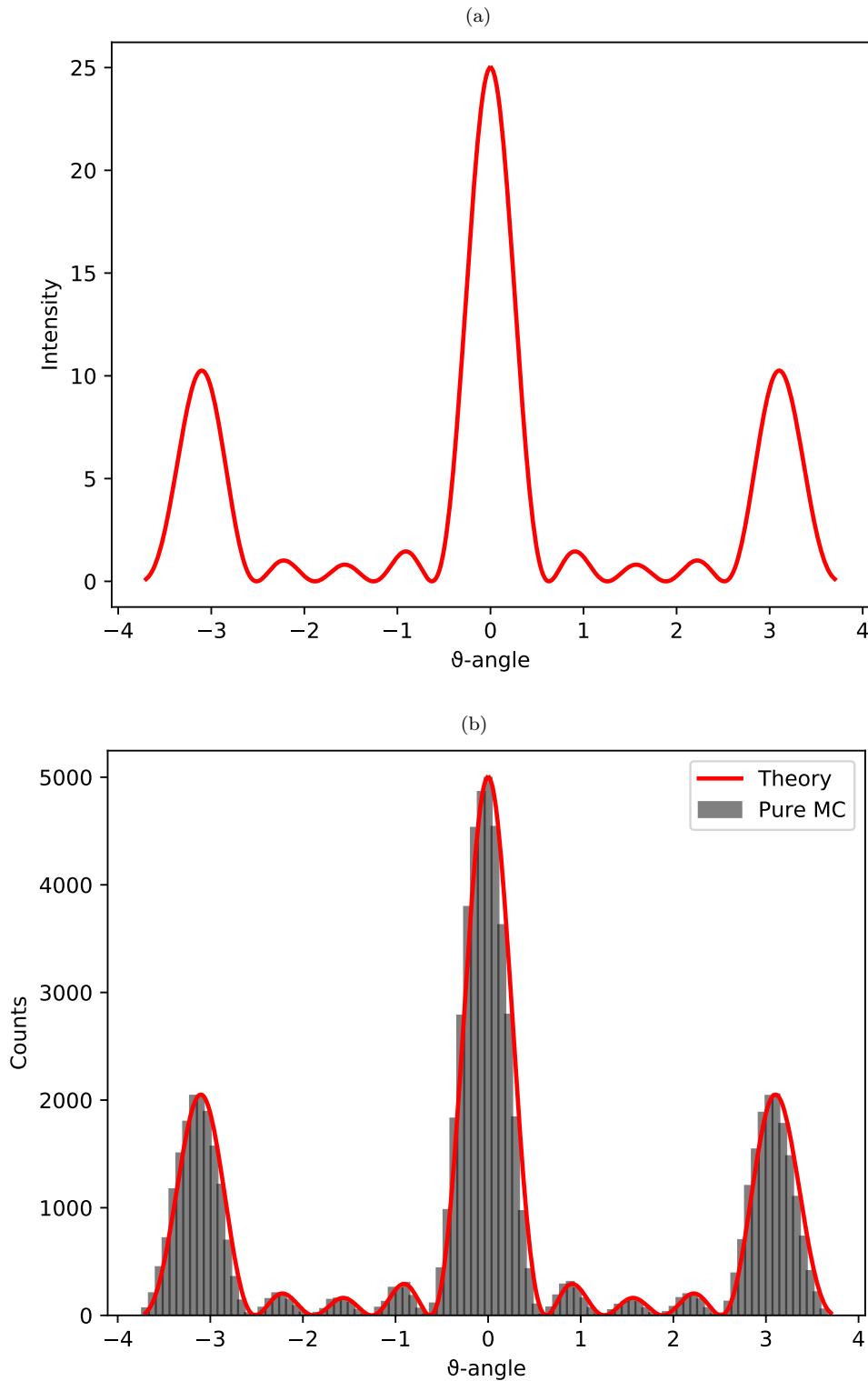
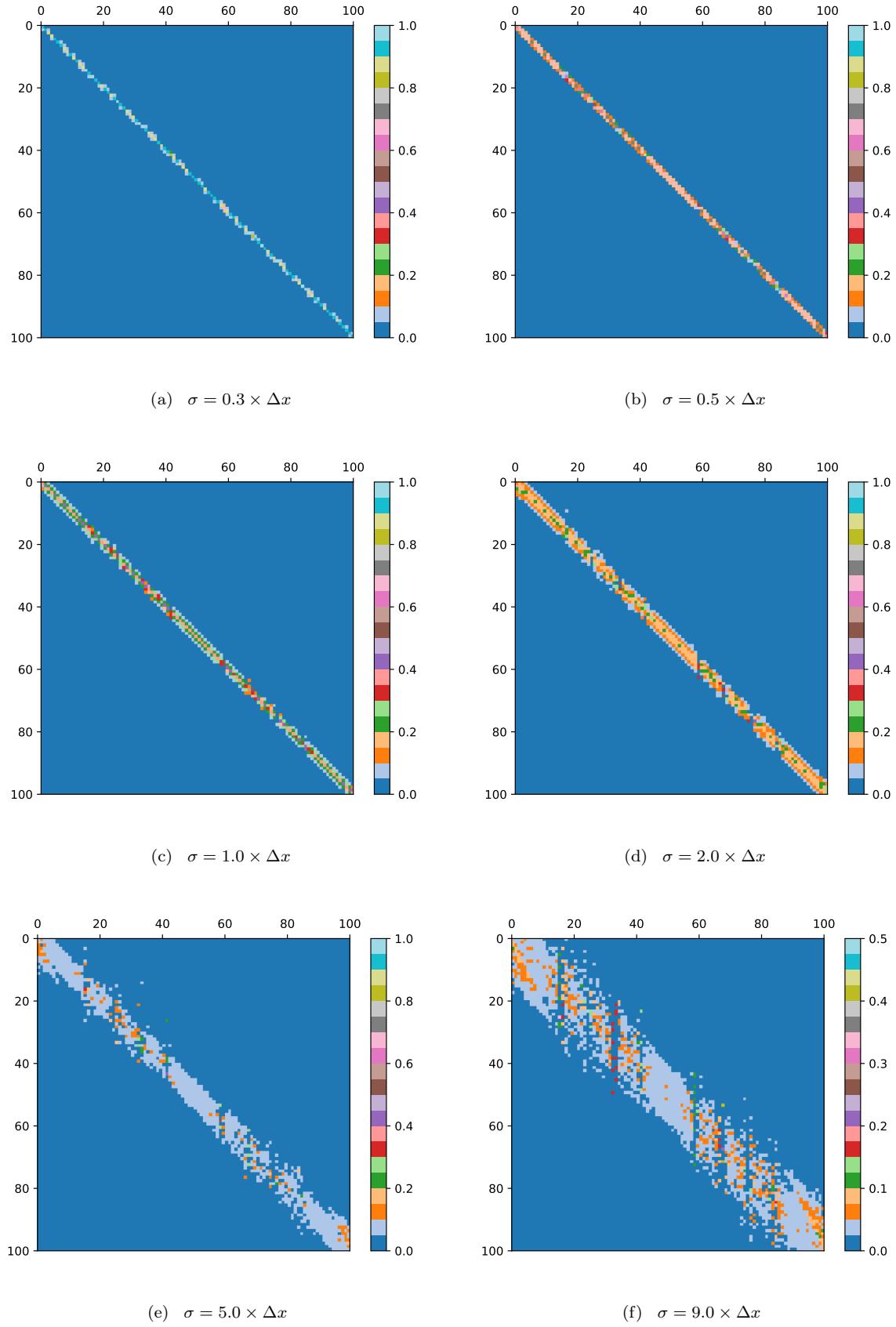


Figura 35: Rappresentazioni in *colormap* delle matrici di smearing generate per diversi valori di σ .



In Figura 36 sono riportati gli istogrammi relativi ai diversi livelli di smearing operato sulla simulazione Monte-carlo "pura". Osserviamo che al crescere del parametro di smearing c si ha una maggiore migrazione tra i bin, con il progressivo svuotamento dei picchi fino alla totale sparizione di quelli minori, coperti dal fondo prodotto dalle code di quelli principali. Tali risultati ben corrispondono a ciò che ci si aspetta da una misura spettrale poco risoluta.

* * *

Resta infine da simulare l'operazione inversa, ossia l'*unfolding* dei dati sperimentali misurati e quindi convoluti con il kernel di risposta del rivelatore. Lo scopo è di ottenere un segnale che riproduca il più possibile il profilo d'intensità incidente sullo schermo, ossia dei dati direttamente confrontabili con il modello teorico da testare. Assumeremo dunque i dati precedentemente sottoposti a smearing come input sperimentale da analizzare e confronteremo il risultato deconvoluto con l'intensità teorica predetta dal modello di Fraunhofer.

Il metodo di unfolding presentato è quello dei cosiddetti "coefficienti di correzione *bin-by-bin*". Si tratta sostanzialmente di definire dei coefficienti correttivi per il valore di ciascun bin, basandosi su delle simulazioni di segnale puro e smussato analoghe a quelle presentate poco sopra. Formalmente, chiamate T_i la frequenza attesa nel bin i -esimo per il segnale a monte della risposta strumentale, e R_i la frequenza attesa per i conteggi effettivamente raccolti nel *pixel* i -esimo del rivelatore (entrambe quantità simulabili per mezzo di generatori pseudo-casuali), definiamo i coefficienti di correzione *locali*²⁹

$$C_i = \frac{T_i}{R_i}, \quad (11)$$

per mezzo dei quali si potrà stimare la distribuzione U_i , deconvoluta a partire dai dati sperimentali D_i , applicando la semplice relazione di proporzione

$$U_i = C_i D_i. \quad (12)$$

Inoltre, se il numero di eventi raccolti su ciascun bin è sufficientemente elevato, a tale stima può essere associata una deviazione standard data da

$$\sigma(U_i) = C_i \sqrt{D_i}. \quad (13)$$

Il predittore U_i ha dunque il pregio di avere una varianza piuttosto ridotta, ma al prezzo di introdurre - perlomeno nel caso generale - un bias sul valor medio dei dati ricostruiti. In particolare è facile dimostrare che

$$\mathcal{B}(U_i) = \left(\frac{T_i}{R_i} \Big|_{\text{MC}} - \frac{\mu_i}{D_i} \right) = C_i^{\text{MC}} - \frac{\mu_i}{D_i}, \quad (14)$$

ossia l'unfolding non introduce bias a patto che la simulazione Montecarlo riproduca in maniera fedele la fisica dell'esperimento e della rivelazione.

I risultati dell'unfolding sono riportati in Figura 37, comparati con il profilo spettrale dato dalla (8). Osserviamo che, anche per i valori più grandi di c , la ricostruzione appare qualitativamente corretta, con ottima sovrapposizione³⁰ tra istogramma dei dati deconvoluti e curva teorica.

²⁹La matrice inversa $\{\{\mathcal{M}^{-1}\}\}$, che se nota ci darebbe l'unfolding esatto dei dati, rappresenta una relazione *non locale* fra i bin, in quanto il valore da assegnare al bin i -esimo dipende in linea di principio dal valore di anche tutti gli altri. Il metodo dei coefficienti di correzione invece definisce per ogni bin un fattore correttivo che dipende solo dal valore misurato sul medesimo bin. Va da sé che tale metodo può essere ritenuto affidabile solo per tassi di migrazione inter-bin estremamente ridotti.

³⁰Resta naturalmente il piccolo bias rigido che avevamo notato già sulla simulazione Montecarlo "pura".

Figura 36: Confronto tra segnale "vero" e simulazioni di segnale misurato, ottenute per smussamento gaussiano, con deviazione standard via via crescente.

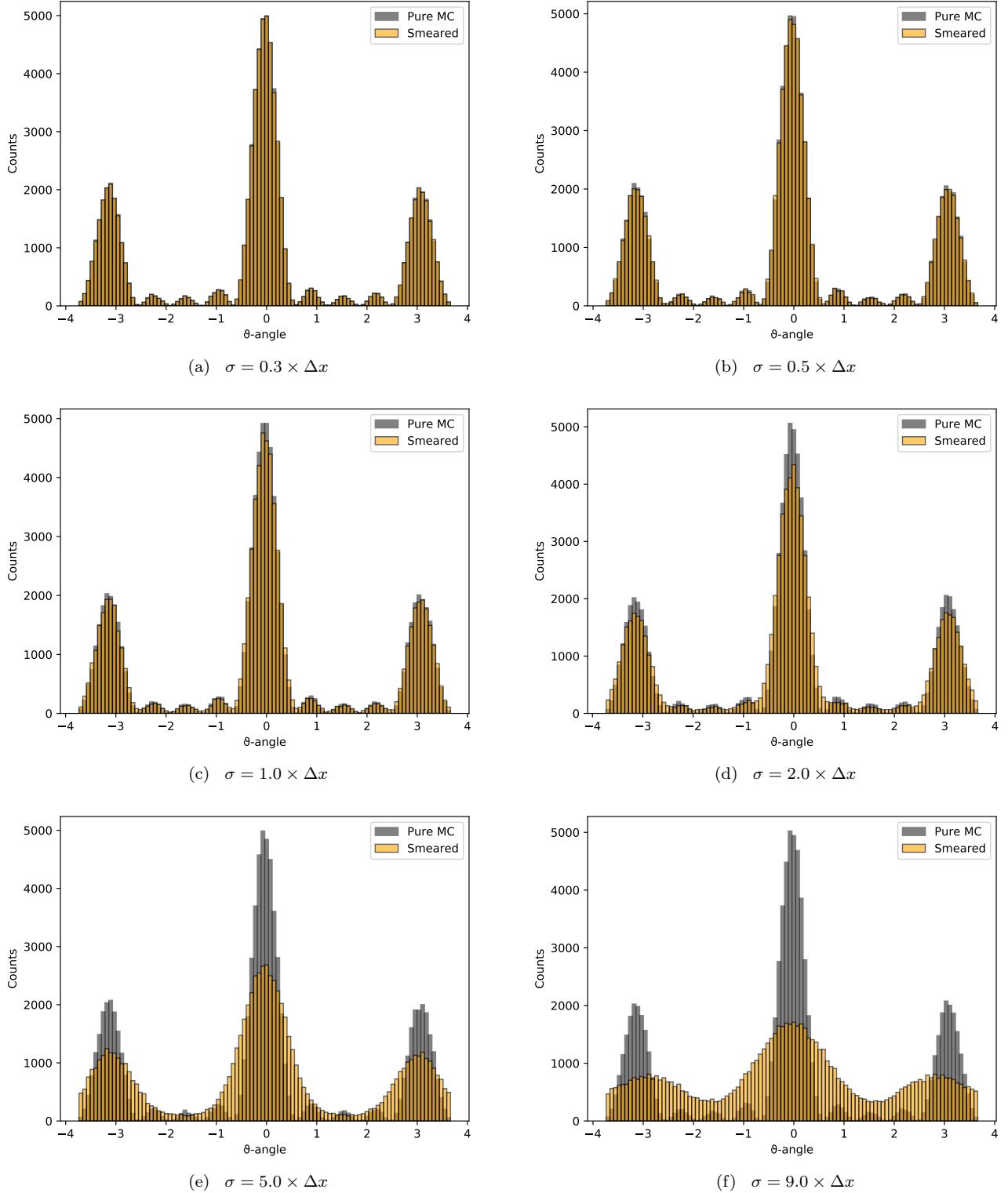
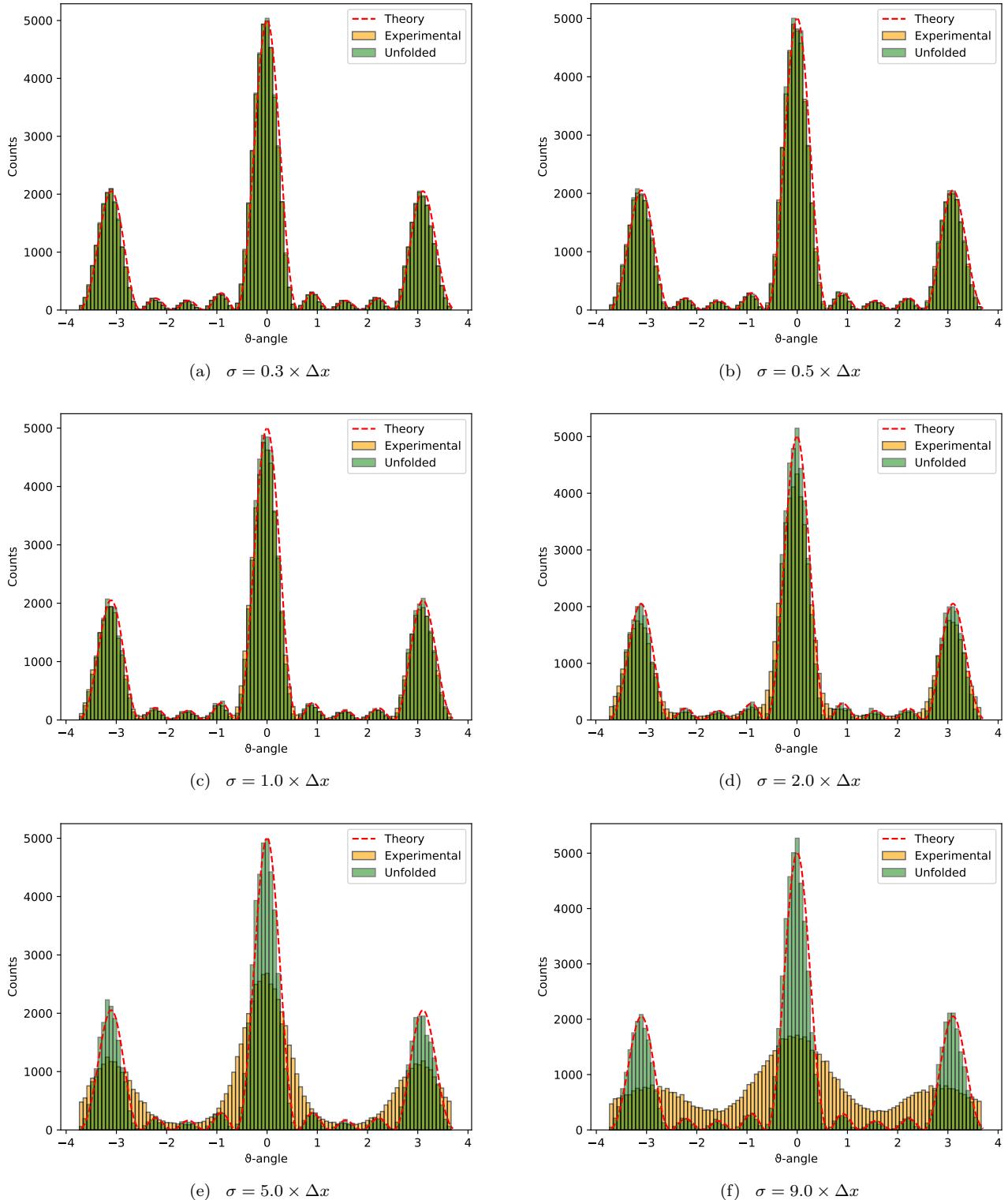


Figura 37: Unfolding delle distribuzioni sperimentali simulate tramite smearing gaussiano di varianza σ^2 e confronto con il profilo d'intensità teorico.



Sono stati inoltre eseguiti dei test d'ipotesi alla Pearson (Cfr. pag.12) per valutare quantitativamente la compatibilità statistica tra i dati ricostruiti per unfolding e la relativa distribuzione Montecarlo "pura" (che nella nostra simulazione rappresenta il segnale sperimentale non convoluto con la risposta del rivelatore). I risultati dei test, riportati di seguito in tabella, hanno evidenziato come in realtà la qualità dell'unfolding decresca fortemente con l'aumentare del parametro di smearing c , come del resto inevitabilmente ci si aspetta.

Smearing Parameter	Calculated χ^2	Pearson's Test $p\text{-value}$
$\sigma = 0.3 \times \Delta x$	5.836227590882934	$9.999999999999999 \times 10^{-1}$
$\sigma = 0.5 \times \Delta x$	46.07973100467730	$9.9999884304470930 \times 10^{-1}$
$\sigma = 1.0 \times \Delta x$	125.5552504111569	$3.6953389551196786 \times 10^{-2}$
$\sigma = 2.0 \times \Delta x$	163.9468296504027	$4.4389667615274120 \times 10^{-5}$
$\sigma = 5.0 \times \Delta x$	223.2895908776605	$1.374244096699173 \times 10^{-11}$
$\sigma = 9.0 \times \Delta x$	225.4725538157693	$7.343546495541846 \times 10^{-12}$

Per concludere riportiamo per intero il codice utilizzato per la generazione, lo smearing e l'unfolding dei dati discussi:

```

1  """ Standard Libraries """
2  import numpy as np
3  from pylab import *
4  from math import *
5
6  """ Pearson's Test Library """
7  from scipy.stats import chisquare
8
9  ## Fraunhofer Theory for Diffraction [N: #{apertures}]
10 def f(x):
11
12     N = 5
13     I0 = 1
14     a = 1
15     b = 0.5
16     return I0*(np.sin(b*x)/(b*x))**2*np.sin(N*a*x)**2/np.sin(a*x)**2
17
18 ## Smearing Parameter Choice [0 < c < 10]
19 c = 1
20
21 ## First MC-simulation ['hit or miss'] to compute N_true
22 x1 = np.linspace(-3.7,-0.01,200) # Have to decompose the domain to
23 x2 = np.linspace(0.01,3.7,200)    # eliminate the x = 0 singularity!
24 v = max(f(x1))
25
26 N_true = np.array([0])
27 while(sum(N_true==0)>=1): # Have to assure no bin results empty
28
29     u = np.random.uniform(-3.7,3.7,500000)
30     t = f(u)
31     y = np.random.uniform(0,v,500000)
32     boolean = y<t
33     u = u[boolean]
34     N_true,bin_edges1 = np.histogram(u,bins=100) # Pure Theoretical MC-Simulation
35

```

```

36 ## Plotting 'Theory' and 'Theory Vs MC-Simulation'
37 half_bin = 0.5*(bin_edges1[len(bin_edges1)-1]-bin_edges1[len(bin_edges1)-2])
38 bin_values = bin_edges1[:-1]+half_bin
39 half_bin = 0.5*(bin_values[len(bin_values)-1]-bin_values[len(bin_values)-2])
40 bin_edges1 = np.append(bin_values-half_bin,bin_values[len(bin_values)-1]+half_bin)
41
42 plt.figure('Fraunhofer Theory')
43 plt.plot(x1,f(x1), color='r', linewidth=2)
44 plt.plot(x2,f(x2), color='r', linewidth=2)
45 plt.xlabel("$\vartheta$"-angle')
46 plt.ylabel('Intensity')
47
48 plt.figure('TheoryVsMontecarlo')
49 plt.plot(x1,f(x1)*(2*3.7/half_bin), color='r', linewidth=2,label='Theory')
50 plt.plot(x2,f(x2)*(2*3.7/half_bin), color='r', linewidth=2)
51 plt.bar(bin_edges1[:-1],N_true,width=half_bin*2,color='k',alpha=0.5,edgecolor='k',label='Pure MC')
52 plt.xlabel("$\vartheta$"-angle')
53 plt.ylabel('Counts')
54 plt.legend()
55
56 ## Computing Smearing Matrix from Normal-Random-Generator [...from numpy library]
57 n = len(bin_values)
58 delta_x = half_bin*2
59 sigma = c*delta_x
60
61 M = np.zeros((n,n))
62 for j in range(0,n):
63     if N_true[j] == 0:
64         print('error')
65     p = np.random.normal(bin_values[j],sigma,N_true[j])
66     for k in range(0,len(p)):
67         if p[k] < -3.7:          #
68             p[k] = p[k]+2*(bin_values[j]-p[k]) # Folding the tails in...
69         if p[k] > 3.7:           #
70             p[k] = p[k]-2*(p[k]-bin_values[j]) #
71     for i in range(0,n):
72         M[i,j] = sum((p>bin_values[i]-half_bin) & (p<bin_values[i]+half_bin))/(N_true[j]*1.0)
73
74 N_smeared = np.matmul(M,N_true) # ROWxCOL product by numpy package
75
76 T = N_true      # For subsequent use
77 R = N_smeared   # -----
78
79 ## Plotting Smearing Matrix [colormap]
80 fig, ax = subplots()
81 im = imshow(M, cmap=cm.tab20, vmin=M.min(), vmax=M.max(), extent=[0, 100, 100, 0])
82 cb = fig.colorbar(im)
83 ax.xaxis.tick_top()
84
85 ## Second MC-simulation ['hit or miss'] to compute a new, independent, N_smeared [D != R]
86 N_true = np.array([0])
87 while(sum(N_true==0)>=1): # Have to assure no bin result empty
88
89     u = np.random.uniform(-3.7,3.7,500000)
90     t = f(u)
91     y = np.random.uniform(0,v,500000)
92     boolean = y<t
93     u = u[boolean]

```

```

94 N_true,bin_edges1 = np.histogram(u,bins=100) # Pure MC
95
96 n = len(bin_values)
97 delta_x = half_bin*2
98 sigma = c*delta_x
99
100 N_smeared = np.matmul(M,N_true) # Represents a 'Simulated Experiment' to unfold...
101
102 ## Plotting 'True Signal Vs Detected Signal',
103 plt.figure('TrueVsSmeared')
104 plt.bar(bin_edges1[:-1],N_true,width=half_bin*2,color='k',alpha=0.5,edgecolor='k',label='Pure MC')
105 plt.bar(bin_edges1[:-1],N_smeared,width=half_bin*2,color='orange',alpha=0.6,edgecolor='k',label='Smeared')
106 plt.xlabel("$\vartheta$-angle")
107 plt.ylabel('Counts')
108 plt.legend()
109
110 D = N_smeared # For subsequent use
111
112 ## 'Bin-By-Bin Correction Factors' and Unfolding
113 C = T/R
114 U = C*D
115 sigma_U = C*D**0.5
116 dev = abs(U-T)
117
118 for i in range(U.shape[0]):          #
119     if dev[i] > 2.3548200*sigma_U[i]: # Checking error bars...
120         print('Unfolding FAILED!')      #
121
122 N_unfold = U
123
124 ## Chi-square test [Pearson] with respect to the last pure simulation
125 chi,p = chisquare(N_unfold,N_true)
126 print('chi: ', chi)
127 print('p-value: ', p)
128
129 ## Plotting 'Theory Vs Experimental Vs Unfolded Signal'
130 plt.figure('Unfolded')
131 plt.plot(x1,f(x1)*(2*3.7/half_bin), color='r', linewidth=1.5, linestyle='--',label='Theory')
132 plt.plot(x2,f(x2)*(2*3.7/half_bin), color='r', linewidth=1.5, linestyle='--')
133 plt.bar(bin_edges1[:-1],N_smeared,width=half_bin*2,color='orange',alpha=0.6,edgecolor='k',label='Experimental')
134 plt.bar(bin_edges1[:-1],N_unfold,width=half_bin*2,color='g',alpha=0.5,edgecolor='k',label='Unfolded')
135 plt.xlabel("$\vartheta$-angle")
136 plt.ylabel('Counts')
137 plt.legend()
138
139 ## Showing all plots
140 plt.show()

```

Listing 14: Python code for Exercise 6 (Smearing and Unfolding routines)

Esercizio 7

Si generino due classi di eventi. La prima classe chiamata *segna* con distribuzione gaussiana bidimensionale centrata in $(x, y) = (4, 4)$ e parametri $\sigma_x = \sigma_y = 1$, $\rho = 0$; la seconda chiamata *fondo* con distribuzione normale bidimensionale centrata in $(x, y) = (0, 1)$ e parametri $\sigma_x = \sigma_y = 1.5$, $\rho = 0.7$. Si utilizzi uno dei metodi MVA illustrati a lezione per separare le due classi di eventi e si caratterizzi il risultato in termini di purezza del segnale e rigezione del fondo.

* * *

In generale, considerate due classi di eventi multivariati \vec{x}_0 , $\vec{x}_1 \in \mathbb{R}^n$ corrispondenti alle rispettive ipotesi H_0 : "Evento da attribuire al fondo" e H_1 : "Evento da attribuire al segnale", si può pensare di definire dei test d'ipotesi in grado di attribuire ogni singolo evento a una classe o all'altra, lavorando su una variabile scalare opportunamente costruita a partire da \vec{x}_0 e \vec{x}_1 . In tal modo da un lato si riduce convenientemente la dimensionalità del problema (rispetto al definire un *decision boundary* $n - 1$ dimensionale) e allo stesso tempo, se facciamo in modo di combinare additivamente le differenze tra H_0 e H_1 , si massimizza l'efficacia del test d'ipotesi.

Dal momento che nel nostro caso trattiamo due distribuzioni gaussiane con media diversa possiamo semplicemente definire un *decision boundary* lineare e proiettare i dati lungo una retta perpendicolare ad esso. Su tale proiezione si effettuerà il taglio per il test d'ipotesi.

L'idea è quindi di definire uno scalare $t(\vec{x}) = \vec{a} \cdot \vec{x}$, dove il parametro $\vec{a} \in \mathbb{R}^n$ sia scelto in modo che la separazione tra le distribuzioni teoriche $g(t|H_0)$ e $g(t|H_1)$ sia massima. Per fare questo si massimizza la quantità

$$J(\vec{a}) = \frac{(\tau_1 - \tau_0)^2}{\sigma_0^2 + \sigma_1^2},$$

dove τ_0 e τ_1 indicano i valori medi della variabile t rispettivamente per fondo e segnale e similmente le σ_0^2 , σ_1^2 ne rappresentano le varianze. La $J(\vec{a})$ rappresenta una buona misura della "distanza" tra le due distribuzioni per $t(\vec{x})$, dal momento che rapporta il bias tra i due valori medi alla somma delle rispettive varianze.

Dalla condizione $\partial_{\vec{a}}[J(\vec{a})] = 0$ si ottiene $\vec{a}_{\max} \propto \mathbb{W}^{-1}(\vec{\mu}_1 - \vec{\mu}_0)$, con $\vec{\mu}_0$ e $\vec{\mu}_1$ valori medi rispettivamente di \vec{x}_0 e \vec{x}_1 , e \mathbb{W} matrice di covarianza combinata, per cui in sostanza dobbiamo proiettare i dati generati lungo la retta individuata da \vec{a}_{\max} e su tale proiezione fissare un t_{cut} per il test d'ipotesi. La statistica $t(\vec{x})$ è nota come *discriminante lineare di Fisher*.

L'implementazione di tale metodo sugli eventi generati risulta dunque nel calcolo delle medie campionarie \vec{m}_i e della conseguente matrice di dispersione combinata $\mathbb{S}_w = \mathbb{S}_0 + \mathbb{S}_1$, con $\mathbb{S}_i = \sum_{\vec{x}} (\vec{x} - \vec{m}_i) \cdot (\vec{x} - \vec{m}_i)$; quindi nella costruzione del discriminante campionario come la proiezione dei dati sul vettore

$$\vec{v} = \mathbb{S}_w^{-1}(\vec{m}_1 - \vec{m}_0).$$

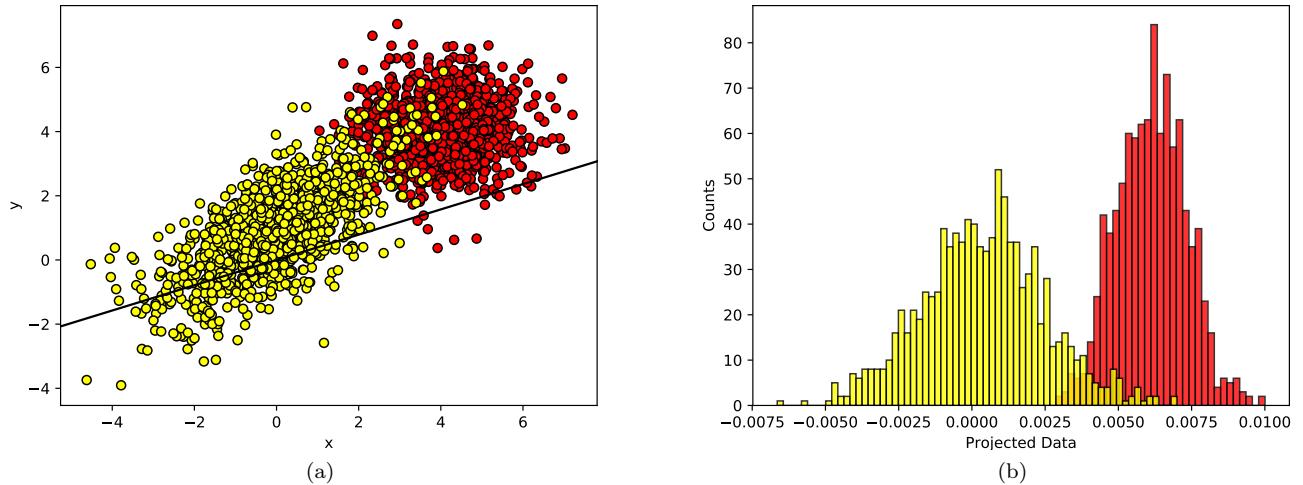
Riportiamo di seguito il codice per la generazione di 10^3 eventi per classe e la costruzione del relativo discriminante lineare:

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 import math
4 import matplotlib.mlab as mlab
5 import matplotlib.lines as mlines
6 import scipy.stats as stats
7
8 ## Background Class H0 Definition
9
10 mean0=[0,1]
11 sigmax0=1.5
12 sigmay0=1.5
13 rho=0.7
14 cov0=[[sigmax0**2,rho*sigmax0*sigmay0],[rho*sigmax0*sigmay0,sigmay0**2]]
15 n0=1000
16
17 ## Signal Class H1 Definition
18
19 mean1=[4,4]
20 sigmax1=1
21 sigmay1=1
22 cov1=[[sigmax1**2,0],[0,sigmay1**2]]
23 n1=1000
24
25 ## Pseudo-Random generation for H0 e H1 gaussians with defined parameters
26
27 x0,y0=np.random.multivariate_normal(mean0,cov0,n0).T
28 T0=np.column_stack((x0,y0))*1.0 ## Data on two columns
29 m0=np.sum(T0, axis=0)/n0 ## Mean over rows
30
31 x1,y1=np.random.multivariate_normal(mean1,cov1,n1).T
32 T1=np.column_stack((x1,y1))*1.0 ## Data on two columns
33 m1=np.sum(T1, axis=0)/n1 ## Mean over rows
34
35 ## Covariance matrices calculation
36
37 S_w=(n1)*np.cov(T1, rowvar=False, bias=1)+(n0)*np.cov(T0, rowvar=False, bias=1) ## S_w=S0+S1
38 S_inv_w=np.linalg.inv(S_w) ## S_w inversion with linear algebra numpy library
39
40 ## Fisher vector definition
41
42 v=np.matmul(S_inv_w, m1-m0) ## ROWxCOL product by numpy package
43
44 ## Data projection on \vec{v}
45
46 x11=np.matmul(v, T1.transpose()) ## Signal data projection
47 x00=np.matmul(v, T0.transpose()) ## Background data projection
48
49 ## Plotting Data, \vec{v} and projections
50
51 plt.figure('Data and v')
52
53 plt.scatter(x1,y1,color='red', edgecolor='black')
54 plt.scatter(x0,y0, color='yellow', edgecolor='black')
55
56 def newline(p1, p2): ## Straight line from two given points...

```

Figura 38: (a) Scatter-plot degli eventi generati con indicata la retta individuata dal vettore \vec{v} e (b) istogramma della proiezione dei dati su tale retta. In rosso gli eventi generati come segnale e in giallo quelli generati come fondo.



```

57 ax = plt.gca()
58 xmin, xmax = ax.get_xbound()
59 if(p2[0] == p1[0]):
60     xmin = xmax = p1[0]
61     ymin, ymax = ax.get_ybound()
62 else:
63     ymax = p1[1]+(p2[1]-p1[1])/(p2[0]-p1[0])*(xmax-p1[0])
64     ymin = p1[1]+(p2[1]-p1[1])/(p2[0]-p1[0])*(xmin-p1[0])
65 l = mlines.Line2D([xmin,xmax], [ymin,ymax], color='black', linestyle='--')
66 ax.add_line(l)
67 return l
68
69 newline([0,0], v)
70
71 plt.figure('Data projections') ## (bins choosen for graphic-only purposes)
72 plt.hist(x11, bins=33, color='red', edgecolor='black')
73 plt.hist(x00, bins=66, color='yellow', edgecolor='black')
74
75 ## Showing plots
76
77 plt.show()

```

Listing 15: Python code for Fisher's linear discriminant construction

Osservando i risultati (Fig. 38) appare subito evidente come il metodo di Fisher selezioni la proiezione lineare dei dati che meglio separa le due classi di eventi.

A questo punto procediamo a identificare t_{cut} come il valore di $t(\vec{x})$ che massimizzi combinatamente l'efficienza di segnale (\mathcal{E}_1 : probabilità di accettare l'ipotesi H_1 quando l'evento è effettivamente di segnale) e la rigezione del fondo ($1 - \mathcal{E}_0$: probabilità di rifiutare l'ipotesi H_1 quando l'evento è parte del fondo). Osserviamo che da questa condizione discende l'ottimizzazione della *purezza* \mathcal{P} del segnale ricostruito, definita come "*rapporto fra gli eventi attribuiti al segnale, ossia contenuti nella regione di accettazione individuata da t_{cut} , e la somma degli eventi di segnale e di fondo che complessivamente ricadono in tale regione*". Naturalmente ci si aspetta che nel limite $\mathcal{E}_0 \ll \mathcal{E}_1$ si ottenga $\mathcal{P} \simeq 1$.

All'atto pratico facciamo dunque scorrere per step omogenei il *decision boundary* dal valore minimo al valore massimo della $t(\vec{x})$ ottenuta dalla proiezione dei dati generati e - ad ogni step - valutiamo la somma di *falsi positivi* e *falsi negativi* ottenuti³¹. Minimizzando rispetto a questa quantità otteniamo uno stimatore del t_{cut} ideale. Il codice relativo a tale procedura è riportato in dettaglio nel Listato 16 .

Dai risultati dell'analisi compiuta:

```
>>> (executing file "PythonCodeEs7.py")
0.003962 : t_cut optimum value
0.745 : alpha optimum value
0.002 : beta optimum value
0.9922178988326849 : Signal Purity
```

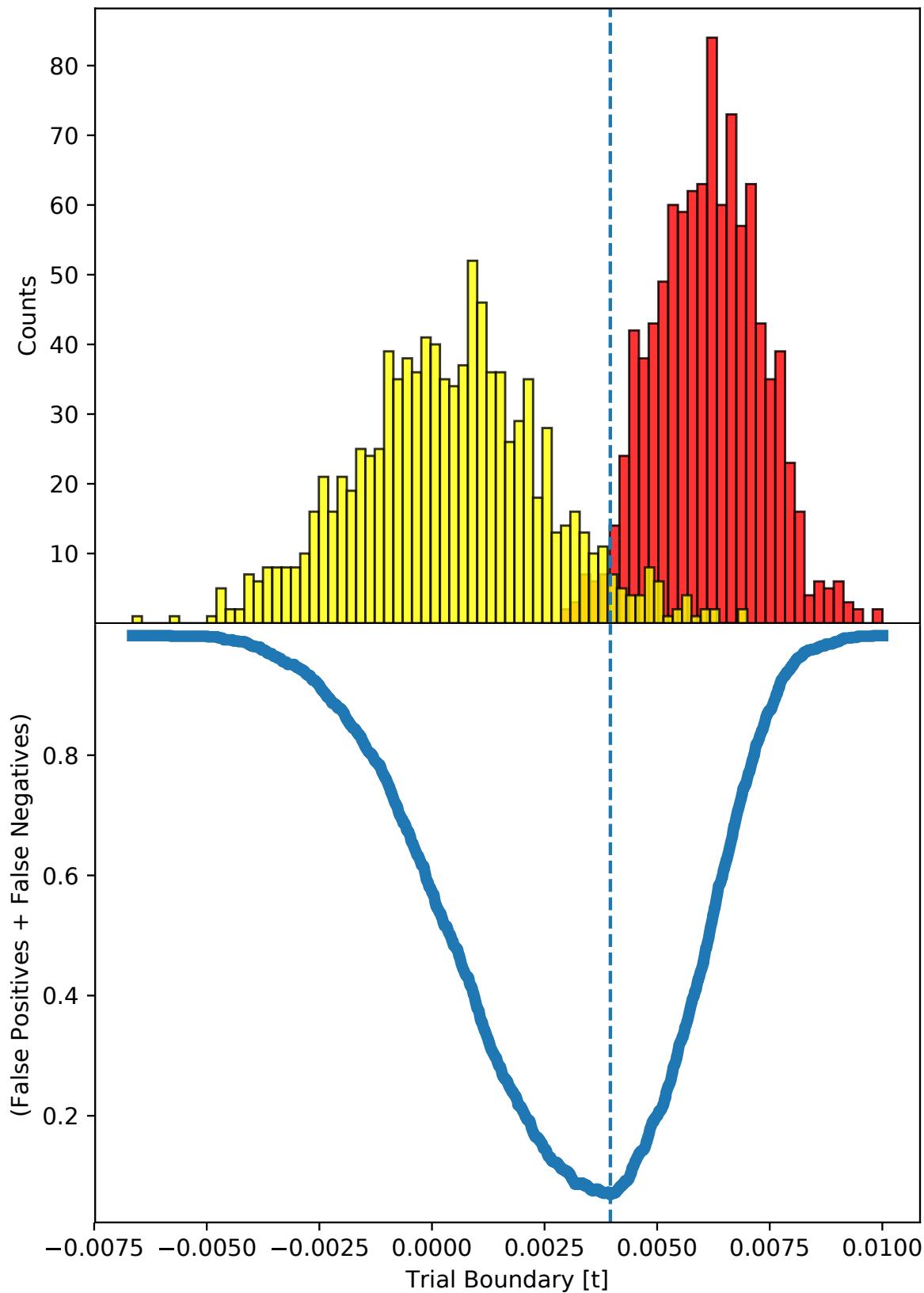
risultano valori coerenti con le richieste poste su \mathcal{E}_0 e \mathcal{E}_1 e una purezza del segnale ottenuta effettivamente molto prossima al valore ideale, segno che l'algoritmo di analisi multivariata implementato risulta efficace per il caso di studio affrontato. In Figura 39 è nuovamente riportato l'istogramma dei dati proiettati, con indicato adesso il *decision boundary* dato dal t_{cut} trovato (tratteggio verticale in blu, corrispondente al minimo di $\alpha + \beta$).

```
1 [...]
2 ## Significance and Specificity optimization and consequent t_cut determination
3
4 t=np.concatenate((x11,x00),axis=0) ## Definition of Fisher Statistic
5 q=np.linspace(min(t),max(t),len(t)) ## Homogeneous domain for optimum-value search
6 alpha=np.zeros(len(t)) ## alpha := 1 - E_1 -> #{False Negatives}
7 beta=np.zeros(len(t)) ## beta := E_0 -----> #{False Positives}
8
9 for i in range(0,len(q)):
10    alpha[i]=(x11<q[i]).sum()/n1 ## Rejection of H1 when H1 is true
11    beta[i]=(x00>q[i]).sum()/n0 ## Acceptance of H1 when H1 is false
12
13 ToMinimize=alpha+beta
14 best=min(ToMinimize)
15
16 t_cut=np.mean(q[ToMinimize==best]) ## There can be more than one best-match...
17 print (round(t_cut,6) , ': t_cut optimum value')
18
19 plt.axvline(t_cut)
20 plt.xlabel('Projected Data')
21 plt.ylabel('Counts')
22
23 ## Plotting alpha+beta as a function of trial t_cut (i.e. q)
24
25 plt.figure('Minimization Functional')
26 plt.plot(q,ToMinimize, linewidth = 5)
27 plt.xlabel('Trial Boundary [t]')
28 plt.ylabel('(False Positives + False Negatives)')
29 plt.axvline(np.mean(q[ToMinimize==best]))
30 plt.show()
31
32 ## Identified signal purity computation
33
34 optalpha=np.mean(alpha[np.where(((q-t)**2)**0.5==np.min((q-t)**2)**0.5)])
35 optbeta=np.mean(beta[np.where(((q-t)**2)**0.5==np.min((q-t)**2)**0.5)])
36
37 print(round(optalpha,6) , ': alpha optimum value')
38 print(round(optbeta,6) , ': beta optimum value')
39 print(round(1-optalpha,6)/(round(optbeta,6)+round(1-optalpha,6)), ': Signal Purity')
```

Listing 16: Python code for Fisher's boundary value optimization

³¹Naturalmente i conteggi di falsi positivi e falsi negativi possono essere messi in relazione con \mathcal{E}_0 e \mathcal{E}_1 rispettivamente nel seguente modo:
 $\beta \sim \mathcal{E}_0$ e $\alpha \sim (1 - \mathcal{E}_1)$.

Figura 39: Ottimizzazione del *decision boundary* sulla statistica $t(\vec{x})$ di Fisher.



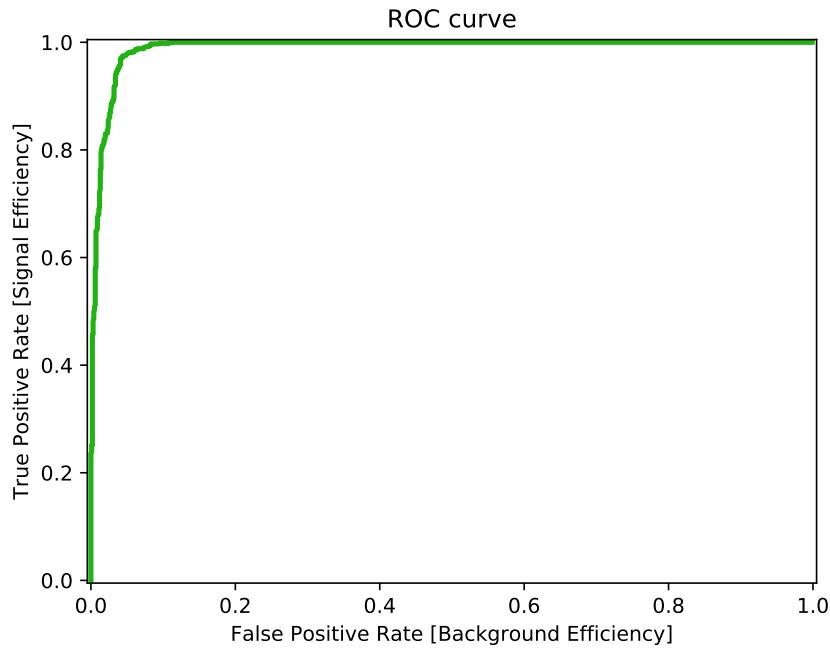


Figura 40: *Receiver Operating Characteristic Analysis for Fisher test*

Infine è stata eseguita un'analisi della curva ROC (*Receiver Operating Characteristic*), la cui area sottesa (AUC) fornisce informazioni sulla qualità del test d'ipotesi implementato: più l'integrale dell'efficienza di segnale valutato come funzione dell'efficienza di fondo ($= 1 - \text{rigezione del fondo}$) è prossimo all'unità e più il test d'ipotesi garantisce di ottenere un segnale ricostruito di grande purezza. Nel nostro caso l'accuratezza con cui è stato approssimato l'integrale (Cfr. Listato 17) non è stata sufficiente a risolvere la differenza $\delta A = (1 - \text{AUC})$, decisamente piccola come evidente in Figura 40.

```

1 [...]
2 ## ROC curve determination
3
4 plt.figure('ROC curve')
5 plt.plot(beta,1-alpha, linewidth = 5, color='green')
6 plt.xlabel('False Positive Rate [Background Efficiency]')
7 plt.ylabel('True Positive Rate [Signal Efficiency]')
8 plt.title('ROC curve')
9 plt.show()
10
11 ## Area under the ROC curve (Integrating in [0,1] with "trapezoid-rule")
12
13 SamplingPts = beta.shape[0]
14 I = 0.5*alpha[0] + 0.5*alpha[beta.shape[0]-1]
15 for j in range(1,SamplingPts):
16     I += alpha[j]
17     dx = beta[j] - beta[j-1]
18     I *= dx
19 I += 1
20
21 print('ROC area: ', I)

```

Listing 17: Python code for ROC curve analysis for Fisher test

Lista dei codici

1	R code for Exercise 1/A (to generate a Breit-Wigner histogram)	5
2	R code for Minimal Standard LCG	10
3	R code for computing non-biased variance and mean predictors	11
4	R code for correlation-study of LCG random sequence	13
5	R code for summing harmonic numbers	14
6	R code for naïve computation of Eulero-Mascheroni constant	15
7	R code for efficient computation of Eulero-Mascheroni constant	17
8	Python code for Simplex Minimization Routine	20
9	Python code for Standard Montecarlo Calculation	38
10	Python code for Importance Sampling Montecarlo Calculation	38
11	Python code for Stratified Sampling Montecarlo Calculation	39
12	Python code for Spectral Peak Recognition Routine	46
13	Python code for Gaussian Smoothing Routine	55
14	Python code for Exercise 6 (Smearing and Unfolding routines)	70
15	Python code for Fisher's linear discriminant construction	74
16	Python code for Fisher's boundary value optimization	76
17	Python code for ROC curve analysis for Fisher test	78

