UNIVERSITÀ DEGLI
STUDI DI FERRARA

Object-Oriented Programming for Experimental Data Analysis

Viola Cavallini
136502
Ingegneria Informatica e dell'Automazione

18/01/2021

Indice

1 Introduzione **2**

2 SiPM **3**

2.1 Analisi in Python . . . 3

2.1.1 analyze_wfs . . . 3

2.1.2 analyze_wfs_anomalies . . . 5

2.1.3 analyze_wfs_no_png . . . 5

2.1.4 Tabella riassuntiva dei tempi . . . 6

2.1.5 plot_peaks . . . 6

2.1.6 calculate_dcr . . . 7

2.1.7 draw_tot_graphs . . . 8

3 Claro **10**

3.1 Ricerca file in Bash . . . 10

3.2 Analisi in Python . . . 10

3.2.1 linear_fit . . . 12

3.2.2 err_fit . . . 12

3.2.3 fit . . . 13

3.2.4 linear_fit_no_png . . . 13

3.2.5 err_fit_for_chips . . . 14

3.2.6 find_chips_threshold . . . 15

3.2.7 sintesi_errori . . . 16

3.3 Analisi in C++ . . . 16

3.3.1 polyfit . . . 16

Capitolo 1

Introduzione

In queste pagine verranno descritti e confrontati 3 script per l'analisi di due diversi set di dati, il primo denominato "SiPM" e il secondo "Claro".

Il set "SiPM" contiene 3 coppie di file *csv*. !!SPIEGAZIONE FILE CSV

Il set "Claro" contiene un numero elevato di brevi file *txt*. Ciascuno di questi file è contenuto in una cartella all'interno di un archivio. L'archivio contiene un elevato numero di cartelle che, create ricorsivamente all'interno di altre cartelle, creano una gerarchia ad albero le cui foglie sono i file *txt*. Il percorso di cartelle dalla radice (archivio) alla foglia permette di trovare il punto di partenza da cui si è originato il segnale i cui valori sono salvati nel file di testo (esempio: al path Station-1–11 -> 2017-10-31-09-46-11-Tray-Station-1–11 -> Chip-001 -> S-curve troviamo 8 file *txt* contenenti i valori registrati dagli 8 canali del chip 001 il 31/10/2017 nella stazione 1-11).

Gli script per leggere ad analizzare i dati sono stati scritti utilizzando Bash, Python oppure C++ e sono stati eseguiti su un computer portatile con processore Intel Core i5.

Capitolo 2

SiPM

2.1 Analisi in Python

Per analizzare il set "Sipm" è stato scritto un solo script Python che si occupa di leggere i file *csv* e disegnare le curve relative alle forme d'onda degli impulsi registrati dai chip.

Python è un linguaggio molto ampio e flessibile in termini di funzioni/pacchetti utilizzabili, permette di scrivere in modo semplice e veloce script relativamente complessi. Questo, a scapito delle prestazioni temporali che ne risentono.

Per questo, si è cercato di ottimizzare il codice creando diversi metodi da usare a seconda del risultato desiderato dall'utente finale, che potrà scegliere se preferire la velocità o la precisione.

Una tabella riassuntiva con i tempi delle funzioni è mostrata in figura 2.3, dopo la descrizione dei singoli metodi.

Oltre la tabella vengono poi descritti i metodi per la stampa su file dei grafici relativi ai valori di picco (funzione *plot_peaks*, per il calcolo della corrente di buio e di altri valori significativi e per la stampa su file dei grafici relativi a questi valori significativi (DCR, CTR, APR).

2.1.1 analyze_wfs

La prima delle 3 funzioni per l'analisi dei picchi è la più lenta, la meno ottimizzata e la più facile da scrivere. Contiene un loop che, per ciascun evento, chiama una funzione di analisi. Questa analizza la curva e restituisce il valore del picco (o dei picchi) e le relative ampiezze. I dati raccolti sono aggiunti al dataframe wf_peaks, che raccoglie tutti i valori dei picchi.

La funzione per l'analisi di un solo impulso, quella chiamata all'interno del loop, sfrutta il pacchetto *Scipy* per trovare i picchi (uno o più) all'interno delle curve. Analizzata la curva, la disegna e la salva come *png*, evidenziando i punti di picco.

Il salvataggio di immagini nel file system locale è estremamente lento. Quando, ad essere salvate, sono migliaia di immagini, diventa eccessivo e spesso non necessario.

Per questo è stata scritta una seconda funzione per l'analisi di un solo impulso, che, invece di disegnare ciascuna curva in un'immagine, salva un file ogni 9 curve, con grafici più piccoli stampati in una griglia 3x3.

In questo modo, i tempi sono ridotti circa del 50%. Niente di eccezionale, ma potrebbe tornare tornare utile nel caso in cui si vogliano tutte le curve disegnate per vederne gli andamenti, ma ci si può accontentare di non avere grafici ampi e precisi.

Un grafico con i tempi si può vedere in figura 2.1, che compara i due andamenti della funzione *analyze_wfs* quando il parametro *compact* è impostato a *True* oppure a *False*. L'asse *x* rappresenta il numero di eventi analizzati. Analizzare 500 eventi con la funzione compatta usa circa lo stesso tempo della funzione originale con 200 eventi.

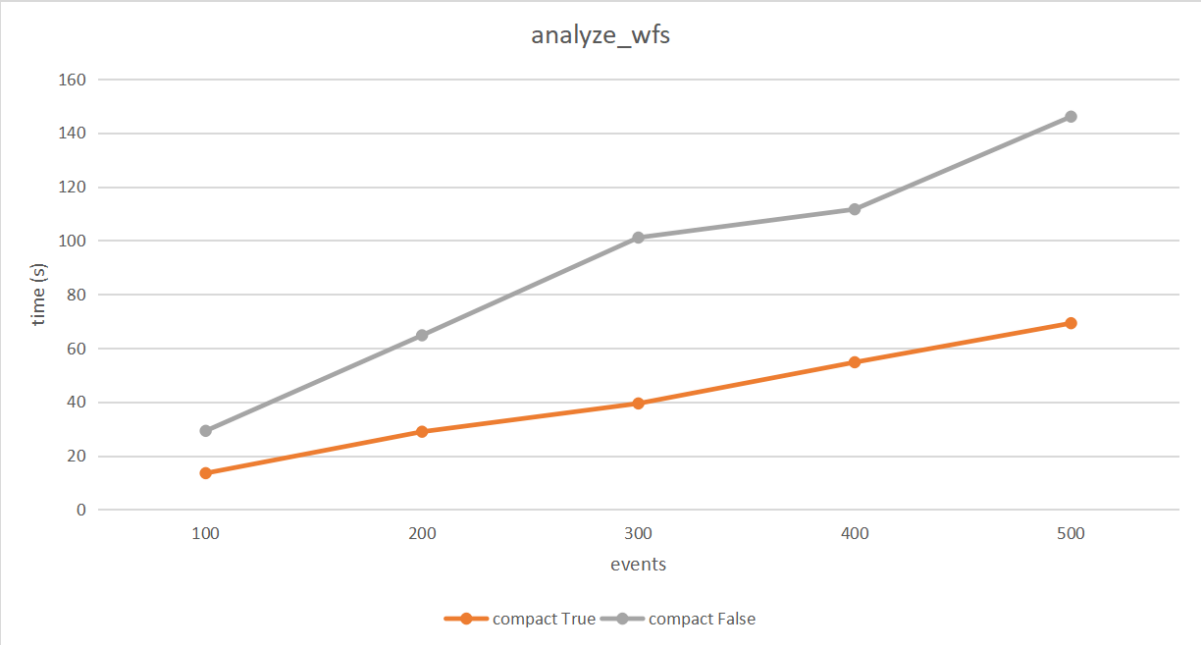

Figura 2.1: I tempi di analyze_wfs, con il parametro 'compact' a True o False

Un'ottimizzazione come questa, sebbene possa risultare utile in alcuni casi, non rende il codice sufficientemente veloce per analizzare in un tempo ragionevole così tanti eventi (in

ciascuna coppia di file ci sono circa 3000 eventi e le coppie sono 3). Per questo sono state create altre 2 versioni della funzione di analisi, descritte di seguito.

2.1.2 analyze_wfs_anomalies

La funzione *analyzie_wfs_annomalies*, come la precedente, analizza tutti i singoli eventi tramite un loop. Siccome è la stampa dei file che rallenta pesantemente il codice, la funzione disegna solamente le curve che ritiene "significative", cioè i "casi particolari" trovati analizzando i singoli eventi. Questi casi includono: eventi con picchi più alti della sensibilità massima del sensore (???), che quindi vengono "appiattiti" e il picco viene trovato a "inf"; eventi con più picchi ravvicinati; eventi con un numero di picchi eccessivamente alto, dovuto ad un andamento anomalo della curva.

Questi eventi "anomali" sono rappresentati in griglie 3x3 per diminuire ulteriormente il tempo necessario alla sua esecuzione.

Un esempio di una griglia 3x3 di eventi "standard" a confronto con una di eventi "anomali" è mostrato in figura 2.2.

Questa funzione analizza 9000 eventi (il dataset completo) in circa 103 secondi, che corrispondono a meno di due minuti.

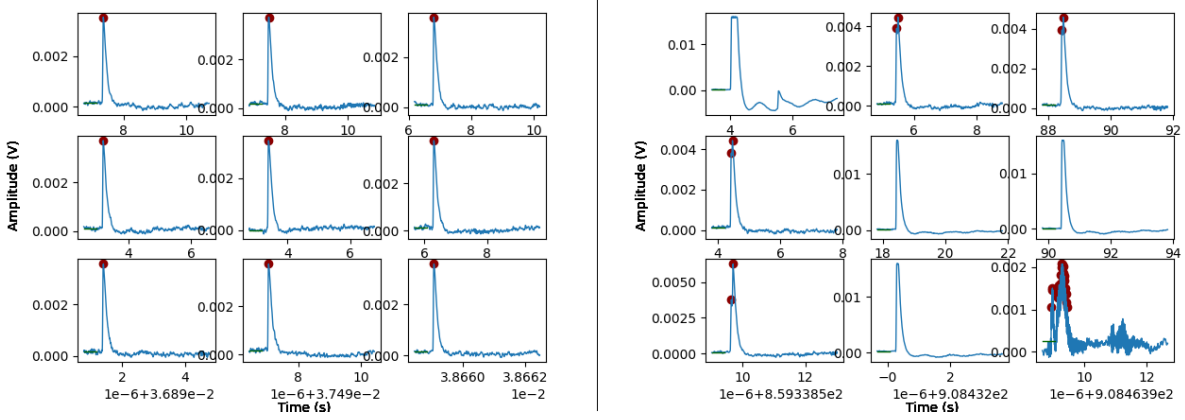

Figura 2.2: Grafici di eventi standard ed eventi "anomali".

2.1.3 analyze_wfs_no_png

L'ultima funzione creata è in assoluto la più veloce: non crea nessuna immagine *png* delle curve. Ovviamente, analizza comunque tutte le curve e salva i valori dei picchi nella variabile

wf_peaks, in modo da poterli usare successivamente.

Questa funzione analizza 9000 eventi (il dataset completo) in circa 23 secondi.

2.1.4 Tabella riassuntiva dei tempi

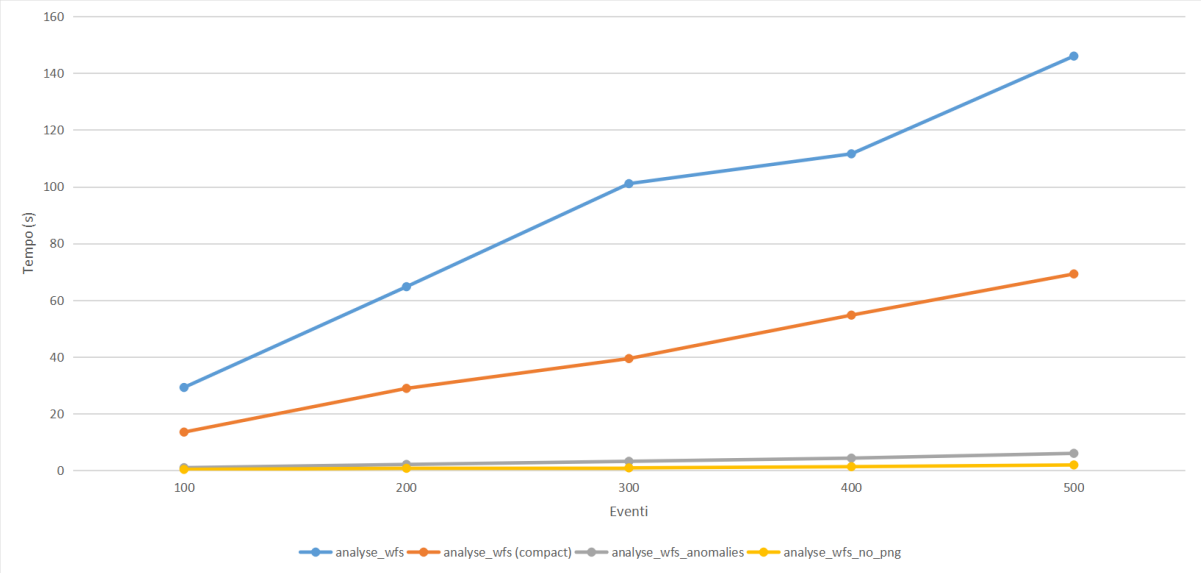

Figura 2.3: I tempi delle varie funzioni di analisi.

2.1.5 plot_peaks

Finita la parte di analisi dei dati, la classe *sipm_wf* ha una struttura Dataframe riempita con i valori di picco, in particolare con le ampiezze e i tempi (dt) relativi ad essi. La funzione *plot_peaks* si occupa di creare grafici significativi che riassumano il comportamento generale dei picchi.

A questo scopo, vengono creati 2 grafici: uno scatter plot e un'istogramma.

Lo scatter plot ha l'ampiezza temporale (dt) del picco sull'asse X e l'ampiezza dell'impulso (A) sull'asse Y.

L'istogramma ha 100 bins rappresentanti 100 intervalli di tempo (o meglio, intervalli di dt) sull'asse X e il numero di picchi appartenenti al relativo bin sull'asse Y.

La figura è salvata in un file *pdf*.

Un esempio di grafico è mostrato in figura 2.4

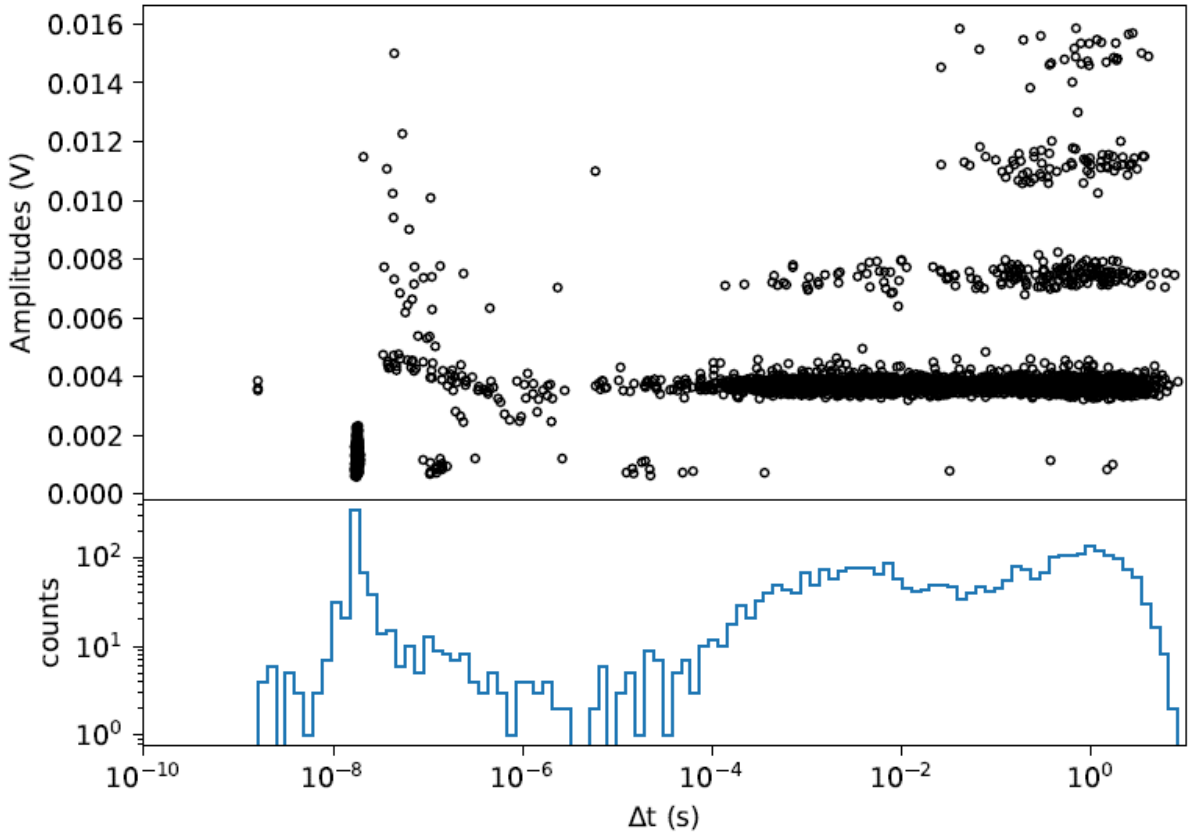

Figura 2.4: Grafico dei picchi ottenuto con la funzione *plot_peaks*.

2.1.6 calculate_dcr

La funzione *calculate_dcr* si occupa di calcolare, a partire dai valori dei picchi, i valori di: corrente di buio, cross-talk e after-pulse. Di ciascuno, viene calcolato il relativo errore (la sua radice quadrata).

La funzione ha, come parametro, una stringa chiamata "group_name". Questa è necessaria per denominare correttamente un file in cui verranno salvati i 3 valori, assieme al valore della corrente di buio, di cui sarà creato un grafico tramite la funzione *draw_tot_graph*.

Non è importante il nome scelto per il gruppo, ma è fondamentale che la stringa sia univoca per il gruppo di dati scelto. In questo caso, siccome le coppie di file che compongono il dataset sono 3, è necessario che la stringa sia la stessa al momento della chiamata del metodo, per ciscuno dei 3 file. Solo così la funzione successiva potrà disegnare in un grafico tutti i 9 valori trovati (DCR, CTR e APR moltiplicati per le 3 coppie di file).

La stringa permette di analizzare le tre coppie di file in tempi diversi (intersecando la loro esecuzione con altri 'gruppi' di coppie di file) e di avere comunque tutti i valori scritti nel file

corretto. In questo dataset, in cui le coppie di file sono soltanto 3, può sembrare inutile, ma nel caso in cui il dataset venga espanso e/o ne vengano creati altri, potrebbe tornare utile avere un nome (e quindi un file dedicato) per ciascun gruppo.

La funzione richiede anche un altro parametro, questa volta obbligatorio: una stringa che contenga il valore numerico della corrente di buio. Questo è stato fatto perché i file di questo dataset sono chiamati, ad esempio, "r203ov.csv", perciò è molto comodo ricavare direttamente il valore di OV dalla stringa, piuttosto che chiederla all'utente. La riga 5 della funzione esegue l'operazione "OV -= 200", questo è necessario per avere un valore di OV corretto in questo dataset. Se il dataset dovesse essere cambiato, questa riga andrà cancellata.

2.1.7 draw_tot_graphs

L'ultima funzione significativa della classe *sipm_wf* è quella che si occupa di disegnare il grafico finale ("totale", da qui il nome) con i valori corrente di buio, cross-talk e after-pulse raccolti, in relazione ad OV.

Il grafico è disegnato a partire dai valori contenuti nel file scritto dal metodo *calculate_dcr*, è quindi fondamentale che la stringa con il nome del gruppo passata come argomento a questa funzione sia la stessa passata all'altra funzione.

Il file *png* generato conterrà il valore di OV sull'asse X e i 3 valori di *rate* sull'asse Y.

Un esempio di grafico si può vedere in figura 2.5. I colori sono uguali per i punti appartenenti alla stessa coppia di file (quindi stesso OV), mentre i *marker* sono uguali se si tratta dello stesso *rate*, come riporta la legenda in alto a sinistra. Le barre degli errori sono, in generale, molto piccole per poter essere viste facilmente, ma sono presenti e si vedono abbastanza bene nei *marker* relativi alla corrente di buio.

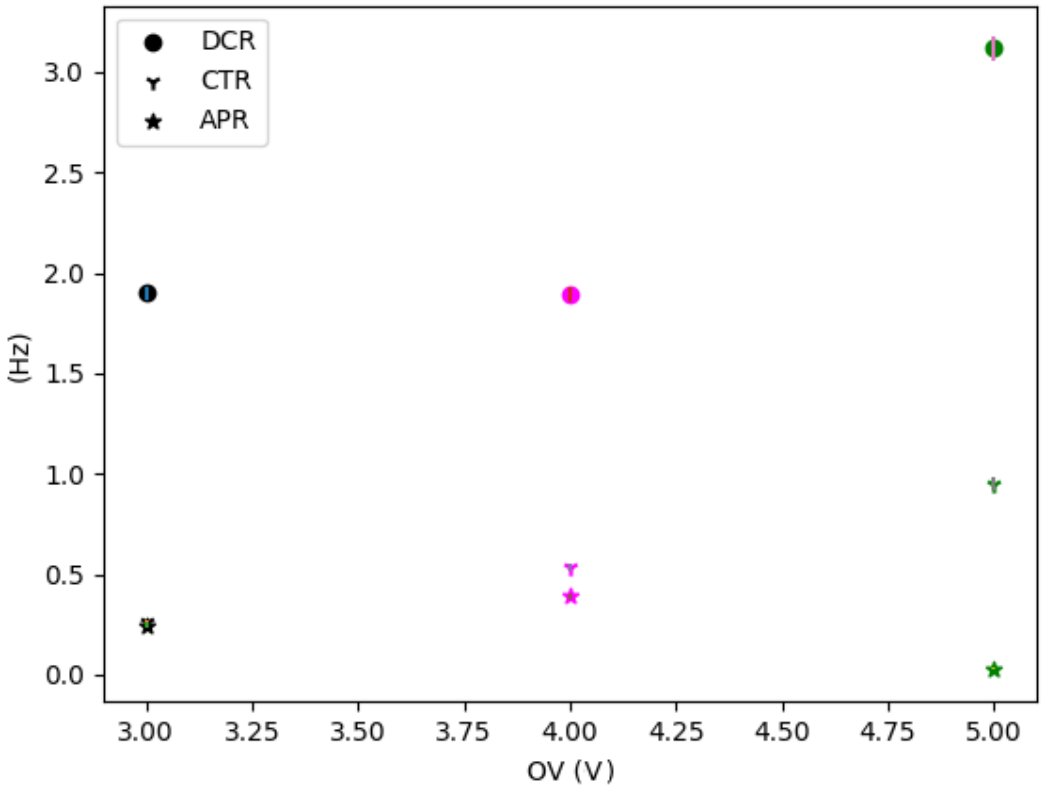

Figura 2.5: DCR, CTR e APR in relazione ad OV.

Capitolo 3

Claro

3.1 Ricerca file in Bash

Il dataset "Claro", come descritto precedentemente, è composto da molteplici file *txt* (circa 41.500) sparsi in una gerarchia complessa di cartelle. Per questo, è stato utile usare il comando bash "find" per creare un file *txt* con tutti i path completi dei file da analizzare.

Il comando è riportato qui di seguito; ad eseguirsi impiega 5.71 secondi, dato misurato con il comando bash "time".

```
find . -path "*/Station_?__??/Station_?__??_Summary/Chip_???/
    S_curve/Ch_?_offset_?_Chip_???.txt" >> file_path.txt;
```

Il comando è salvato in un file bash chiamato "analisi_file.sh", che sarà richiamato dai programmi Python e C++ per creare il file *txt* con i path.

Sicuramente ci sarebbe stato il modo di fare la stessa cosa utilizzando cicli o ricorsioni sia con funzioni Python che C++, in modo da rendere i programmi scritti coerenti. Essendo, però, il bash estremamente ottimizzato per la ricerca ricorsiva all'interno del file system, si è preferito sfruttare questo linguaggio per poter avere più di 40 mila path stampati su file in poco più di 5 secondi.

3.2 Analisi in Python

Come per il dataset "SiPM", anche per "Claro" è stata scritta una classe Python con metodi per l'analisi dei dati.

La classe ha i metodi "base" per calcolare un fit lineare e un fit migliore tramite la funzione *erf()* del modulo Scipy. Entrambi i metodi calcolano il fit e poi disegnano su file *png* sia i

punti che rappresentano i valori acquisiti, sia la retta/curva trovata con i calcoli. Entrambe le funzioni creano un'immagine *png* per ciascuna curva (ovvero per ciascun file), sono quindi estremamente lente.

Oltre alle due singole funzioni, è stata scritta anche la funzione *fit()* che "unisce" i due fit, disegnando in un unico grafico sia i punti, sia il fit lineare, sia quello trovato tramite la funzione *erf()*.

L'ultima tra le funzioni che analizzano i singoli file è *linear_fit_no_png()*, che, come dice il nome, esegue il fit lineare e, invece di salvare immagini, salva i valori su un file testuale. La funzione servirà successivamente per avere un confronto sulle tempistiche del programma Python in confronto a quello C++.

Oltre a queste funzioni più "semplici" e basilari, ne sono state scritte altre 2 per l'analisi più approfondita dei chip: *err_fit_for_chips()* e *find_chip_threshold()* verranno descritte qui di seguito.

La maggior parte delle funzioni accetta 2 argomenti: *how_many* e *how_many_chips*. Il primo indica quanti file analizzare, leggendoli in ordine dal file *file_path.txt*. Il secondo, di default a 0, indica quanti chip analizzare. Se questo parametro è diverso da 0, vengono letti *how_many* file da *file_path.txt*, ma vengono considerati solo quelli provenienti da un chip che abbia numero minore o uguale a *how_many_chips*.

Ad esempio, se alla funzione *linear_fit* vengono passati *how_many=80* e *how_many_chips=2*, verranno lette 80 righe dal file *file_path.txt*, ma verranno salvate solo le 16 immagini relativi agli 8 canali dei chip 1 e 2.

Questo è utile se si vuole analizzare, ad esempio, il comportamento di pochi chip durante tutte le acquisizioni, senza però disegnare tutte le curve (quindi utilizzando un tempo ragionevole).

Ad esempio, il chip 001 è responsabile di 149 acquisizioni su circa 40.000. L'ultima sua acquisizione, però, è la 39.470, ciò significa che se volessimo i plot delle sue acquisizioni, dovremmo disegnare quasi 40.000 grafici, una cosa impossibile in un tempo abbastanza breve.

Impostando il parametro *how_many_chips=1*, otteniamo i soli grafici relativi al chip 001 in circa 40 secondi.

Questo parametro sarà particolarmente utile nella funzione *find_chips_threshold*, descritta in seguito.

3.2.1 linear_fit

La funzione più breve e semplice è la *linear_fit()* che, come dice il nome, trova il fit lineare dei punti letti da ciascun file *txt*. Per ridurre al minimo l'errore, vengono presi solo i punti centrali, quelli compresi tra 5 e 990. Se è trovato un solo punto, l'intervallo viene esteso a 1 e 999, sperando di trovare almeno un altro punto. Nel 90% dei casi questo risolve il problema, ma si hanno casi in cui, esclusi i punti = 0 e quelli = 1000, resta comunque un solo punto. In questo caso il fit non va a buon fine: la figura è comunque disegnata, ma nel conteggio finale dei risultati non è considerato.

Per ogni grafico, viene stampato su di esso il valore dei soglia trovato con il fit, quello "reale" letto dal file *txt* e la loro differenza.

3.2.2 err_fit

Questa è la funzione da chiamare se si desiderano i grafici dei punti e della curva di fit trovata tramite la funzione *special.erf()* del modulo Scipy. Ha gli stessi argomenti di *linear_fit*, ovvero *how_many* e *how_many_chips*.

La curva trovata tramite la funzione *erf()* viene adattata orizzontalmente e verticalmente in modo da ridurre il più possibile la distanza tra i punti che rappresentano i valori.

Un primo adattamento viene fatto traslando la funzione orizzontalmente di un valore X pari alla X del primo punto e ampliandola verticalmente moltiplicandola per 500 (l'intervallo dei punti va da 0 a 1000, quello della funzione degli errori va da -1 a 1, quindi il calcolo è semplice).

Un secondo adattamento orizzontale viene fatto con lo scopo di avvicinarsi il più possibile ai punti centrali: viene calcolato il primo punto "utile", cioè il primo punto tra quelli centrali, e viene ricavata la X della funzione degli errori in quel punto. Traslando la funzione in orizzontale di un valore pari alla differenza tra le due X trovate (quella del punto e quella della funzione), la si fa combaciare con il punto centrale trovato.

Se ci sono più punti centrali (indicativamente con un Y compresa da 50 e 950), ne viene calcolata la media delle X e la funzione viene traslata di un valor medio tra tutti quelli trovati.

Anche qui, come per il fit lineare, viene stampato sul grafico il valore della soglia trovata tramite la funzione, quella "reale" letta dal file *txt* e la loro differenza.

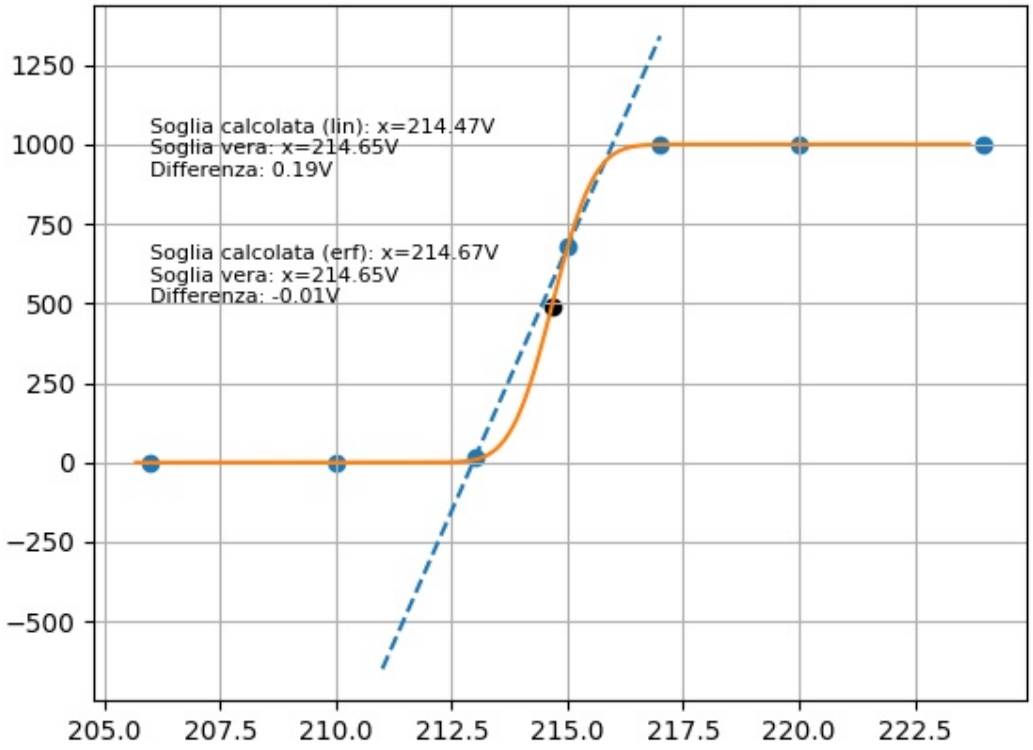

Figura 3.1: Esempio di grafico generato dalla funzione *fit*

3.2.3 fit

La funzione *fit*, come detto precedentemente, unisce le due funzioni di fit, calcolando per ciascun gruppo di punti il fit lineare e quello tramite la funzione degli errori con i metodi descritti per le altre due funzioni e stampando tutto su un unico grafico.

Vengono stampate anche le soglie trovate, quelle "reali" e le rispettive differenze.

Anche questa funzione ha come parametri gli stessi *how_many* e *how_many_chips* delle funzioni precedenti.

Un esempio di grafico creato da questa funzione è mostrato in figura 3.1

3.2.4 linear_fit_no_png

Questa funzione esegue il fit lineare nelle stesse modalità descritte per la funzione *linear_fit*, ma non disegna i grafici, risparmiando così un'enorme quantità di tempo.

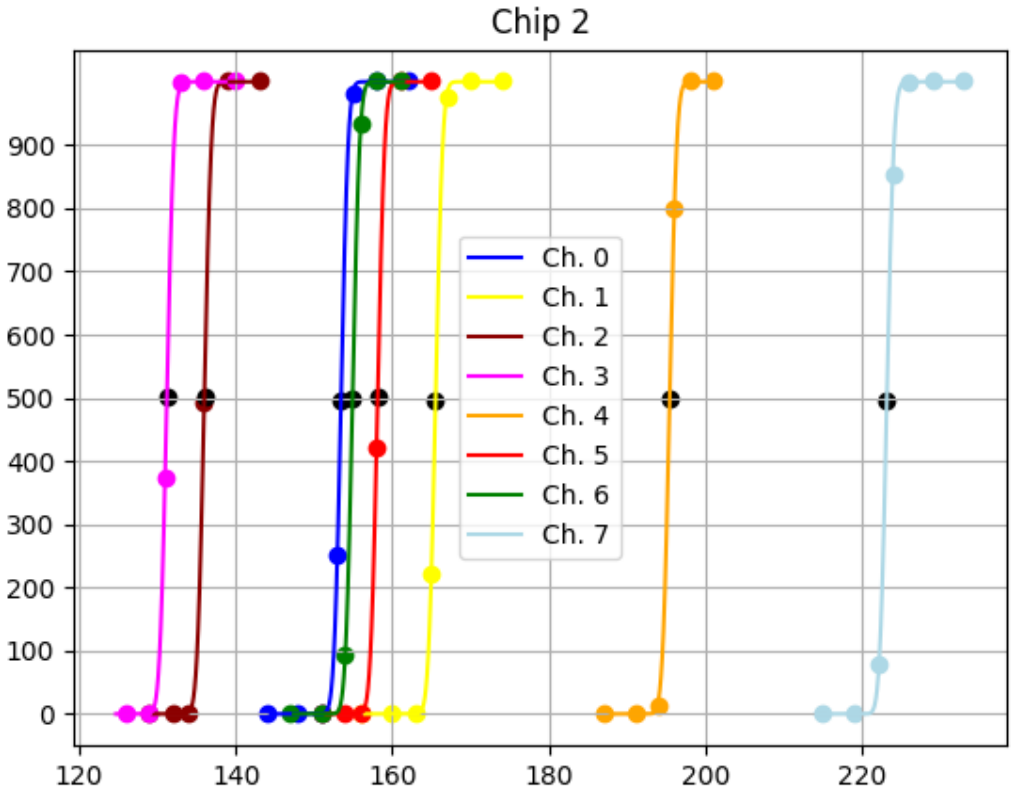

Figura 3.2: Esempio di grafico generato dalla funzione *err_fit_for_chips*

I valori della soglia calcolata, della soglia "reale" letta dal file *txt* e della relativa differenza sono salvati su un file testuale chiamando "risultati_py.txt".

Questa funzione è stata utilizzata per confrontare i tempi tra l'analisi in Python e quella in C++, siccome il programma in C++ si limita a memorizzare i risultati su un file testuale.

3.2.5 err_fit_for_chips

Questa funzione calcola il fit tramite la funzione degli errori (con le modalità descritte in precedenza), poi disegna un grafico per ciascun chip con i punti e i fit degli 8 canali, in 8 colori diversi. Un esempio di grafico è mostrato in figura 3.2 Anche questa funzione, come le precedenti, accetta i parametri *how_many* e *how_many_chips*.

3.2.6 find_chips_threshold

La seconda (e ultima) tra le funzioni per l'analisi del comportamento di ciascun chip, è *find_chips_threshold*. Questa disegna uno scatter plot per ogni chip con i punti rappresentanti le soglie trovate. Le soglie sono trovate tramite fit con la funzione degli errori, in modo da essere il più precise possibili.

In questa funzione è particolarmente importante il parametro *how_many_chips*, che permette di avere grafici relativi a pochi chip ma con molti punti ciascuno (le soglie trovate in acquisizioni diverse vengono sommate, per cui se un chip è responsabile di X acquisizioni, il suo grafico potrà contenere fino a 8*X punti di soglia, indipendentemente dal fatto che le X acquisizioni siano nei primi X file letti o in file distribuiti uniformemente tra le cartelle e quindi letti in momenti diversi).

Per velocizzare ulteriormente la funzione, i grafici sono raggruppati in griglie 3x3. Un esempio di grafico è mostrato in figura 3.3.

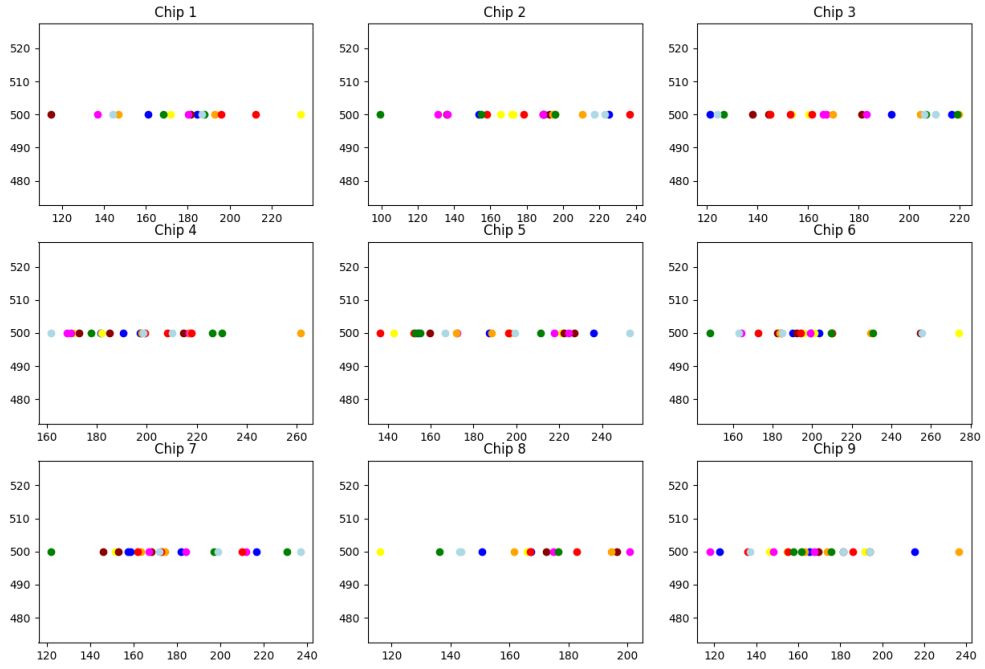

Figura 3.3: Esempio di grafico generato dalla funzione *find_chips_threshold*

3.2.7 sintesi_errori

L'ultima funzione utile all'analisi dati scritta in Python è questa, che semplicemente stampa a video una media degli errori tra le soglie "vere" e quelle trovate con i due tipi di fit.

Affinché stampi a video correttamente le medie, è necessario che venga chiamata dopo le funzioni di fit, in modo che la funzione possa leggere dati corretti (si noti che chiamare *linear_fit* e *err_fit* oppure chiamare solo *fit* è indifferente, siccome i dati analizzati complessivamente sono gli stessi).

Per dare un esempio: con 5000 eventi, l'errore medio per il fit lineare è 0.24, mentre per il fit con funzione degli errori è 0.08.

3.3 Analisi in C++

Lo script per l'analisi in C++ è molto più semplice rispetto a quello in Python: non disegna grafici e analizza i dati facendone solo un fit lineare.

Non dovendo salvare nel file system immagini *png* ed essendo quindi molto più veloce, non ha una variabile con lo stesso scopo del parametro *how_many_chips*, gli si può però ovviamente specificare il numero di file da analizzare.

Il *main()* si compone di un ciclo che legge una riga di *file_path.txt*, apre il file corrispondente, ne legge i valori, seleziona quelli "centrali", li passa alla funzione *polyfit()* e stampa le soglie ottenute nel file *risultati_cpp.txt*.

In figura 3.4 si vedono i tempi dello script in C++ confrontati con quello in Python (che esegue solo la funzione *linear_fit_no_png*).

3.3.1 polyfit

La funzione *polyfit* esegue il fit lineare dei valori X ed Y passati come argomento. Prima viene calcolata la media dei valori X e dei valori Y, da questa è possibile calcolare covarianza e scarto quadratico medio, trovato poi i parametri m e q della retta di regressione lineare.

I risultati, ovviamente, sono quasi identici a quelli trovati con l'analisi in Python (a meno di arrotondamenti).

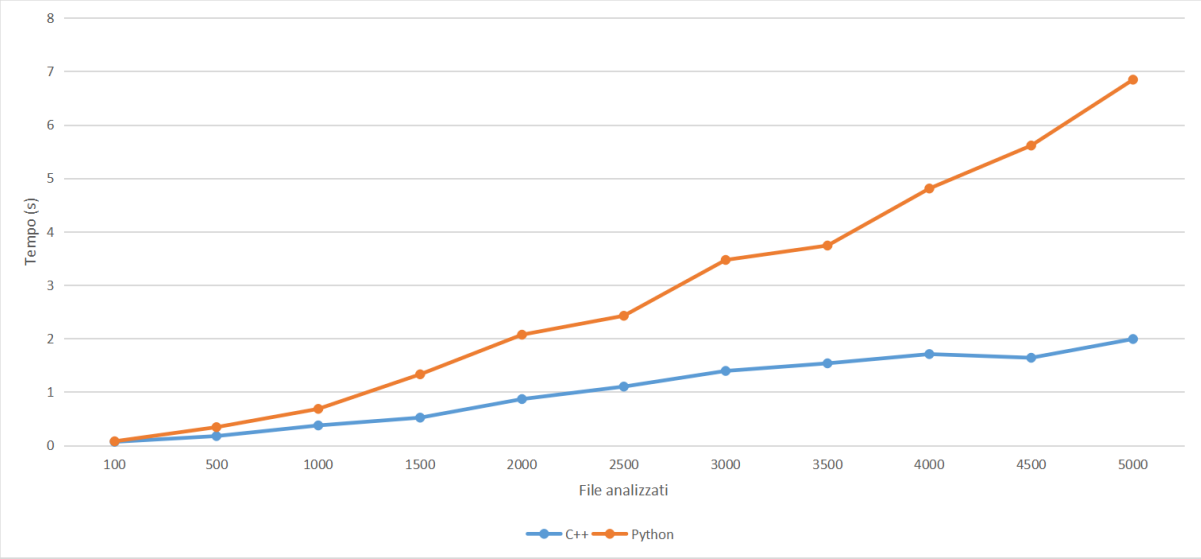

Figura 3.4: Confronto dei tempi tra lo script C++ e quello Python