Arduino e campionamento/digitalizzazione di segnali continui

francesco.fuso@unipi.it; http://www.df.unipi.it/∼fuso/dida

(Dated: version 3 - FF, 7 novembre 2015)

Queste nota ha un duplice scopo: in primo luogo essa presenta alcune generalità sulle modalità di impiego di Arduino tenendo conto di obiettivi, esigenze e limitazioni tipiche delle esperienze di laboratorio. Inoltre essa descrive l'esperienza di misura automatizzata di differenze di potenziale continue (considerate tali) con Arduino e l'esecuzione di istogrammi.

I. INTRODUZIONE

L'esperienza considerata rappresenta una semplicissima ("la più semplice") applicazione di acquisizione automatizzata di dati. In buona sostanza c'è una d.d.p. costante, ovvero supposta tale, prodotta da un partitore di tensione collegato al solito generatore di d.d.p. in uso in laboratorio. Questa d.d.p. deve essere digitalizzata e acquisita un gran numero di volte in istanti successivi allo scopo di creare un array (una colonna) di valori. Dal punto di vista pratico, l'operazione descritta ha poco senso: infatti in genere è utile acquisire, tramite campionamento e digitalizzazione, segnali (d.d.p.) che variano nel tempo, dato che dall'analisi del transiente possono essere dedotte importanti informazioni sui sistemi, o i circuiti, sotto analisi. Tuttavia questa esperienza ha l'indubbia utilità di mostrare le modalità di uso, le potenzialità e le difficoltà connesse all'impiego di un digitalizzatore, in particolare di Arduino [1] usato qui come scheda di acquisizione e conversione analogico/digitale.

Per ottenere i nostri scopi è necessario istruire Arduino a compiere una sequenza di misure della d.d.p., che va collegata a una delle porte analogiche di cui è dotato (nell'esempio è la porta collegata al pin `A0`). Queste misure daranno luogo in modo automatico a un *campione* (contenente centinaia o migliaia di dati) registrato in un file, che può essere analizzato a posteriori anche con tecniche statistiche (media, deviazione standard sperimentale, istogramma delle occorrenze).

Poiché, come chiariremo nel seguito, il trasferimento dei dati da Arduino al computer è generalmente lento (richiede secondi), e che Arduino dispone di una (piccola) memoria interna, l'istruzione impartita ad Arduino prevede che esso immagazzini temporaneamente i risultati delle misure nella sua memoria interna, per poi trasferirli al computer tutti insieme, in "un colpo solo" al termine dell'acquisizione.

II. (NOSTRA) FILOSOFIA DI OPERAZIONE DELLE ESPERIENZE CON ARDUINO

Discutiamo ora alcuni aspetti molto generali che vanno presi in considerazione nel progettare e realizzare esperienze con Arduino. Partiamo con il ricordare che il cuore di Arduino è un microcontroller (modello ATmel ATmega328, per la scheda Arduino Uno), dotato, tra le altre funzioni, di digitalizzatori che possono campionare segnali (d.d.p.) analogici e convertirli in interi con una dinamica di 10 bit, corrispondenti a $2^{10} = 1024$ livelli distinti. Salvo le ulteriori precisazioni che discuteremo in seguito, questi livelli corrispondono all'intervallo di d.d.p. tra (circa) zero e un valore massimo, o di riferimento, tipicamente $V_{ref} \simeq 5$ V. Dunque la sensibilità, o accuratezza, della misura usando la configurazione standard di Arduino è di circa $5/1023 \sim 5$ mV.

Concettualmente il microcontroller riproduce, in forma ridotta e parziale, il processore (CPU) di un qualsiasi computer e quindi come questo ha bisogno in primo luogo di essere istruito sulle operazioni che vogliamo che compia. In un normale computer questo è, grosso modo e senza entrare nei dettagli, quanto viene eseguito dalla combinazione di programma e sistema operativo. Una versione molto semplificata di sistema operativo specifico è residente in una memoria permanente (EEPROM) contenuta nel microcontroller.

Con Arduino le istruzioni di programma possono essere date scrivendo un semplice testo, detto *sketch*, all'interno di un ambiente interattivo, detto IDE, specifico, cioè un programma che si chiama, generalmente, Arduino, Arduino IDE, o Arduino Programming, e che è rilasciato per tutti i principali sistemi operativi. Questo ambiente interattivo è presente nei computer del laboratorio e si individua facilmente essendo identificato dall'icona di Arduino (un infinito su sfondo verde acqua, in genere). Lo sketch, che è composto di parti distinte, tutte funzionali e necessarie, è scritto con una sintassi che ricorda molto da vicino quella del linguaggio C (più precisamente si tratta di una sorta di sotto-insieme del C, implementato nel sotto-insieme di un ambiente/linguaggio che si chiama Processing). Naturalmente lo sketch contiene anche delle istruzioni specifiche per controllare Arduino, per esempio quelle che stabiliscono se le varie porte devono essere considerate come ingressi e uscite, quelle che eseguono la lettura di una porta analogica o la "scrittura" (determinazione di stato "alto" o "basso") per una porta digitale. Fortunatamente, la sintassi di queste istruzioni è abbastanza ben comprensibile e in molti casi addirittura auto-esplicativa.

Il programma Arduino IDE presente sul computer provvede, una volta terminata la redazione ed eseguiti con successo alcuni test preliminari sulla sintassi, a fare l'upload, cioè trasferire con un apposito comando (l'icona è una freccina) il contenuto dello sketch, debitamente

compilato, in una memoria non volatile ("flash") di cui è dotato il microcontroller. Il trasferimento avviene sfruttando la comunicazione seriale USB tra computer e scheda Arduino. Purtroppo le dimensioni della memoria non volatile sono piccoline (32 kB), per cui lo sketch deve necessariamente essere semplice e breve. In particolare, le dimensioni degli array delle variabili sono limitate, con la conseguenza che il numero complessivo di misure che Arduino può fare è limitato. Una volta trasferito lo sketch compilato, le istruzioni diventano *residenti* nel microcontroller, che dunque le eseguirà finché non verrà in qualche modo resettato, per esempio sovrascrivendo lo sketch con uno nuovo. Normalmente nella fase di trasferimento del programma dal computer ad Arduino alcuni led presenti sulla scheda si accendono e si spengono a mostrare che c'è una comunicazione in corso con il computer.

In genere è consigliabile avere una qualche forma di controllo sulla partenza delle operazioni compiute da Arduino (per esempio, nel nostro caso, l'avvio delle misure della d.d.p. presente al pin `A0`). Questo può essere fatto via hardware, per esempio usando un pulsante opportunamente collegato alle porte digitali della scheda. Oppure, come per l'esperienza di cui stiamo trattando, il controllo può essere eseguito via software.

A questo scopo si utilizza ancora la comunicazione seriale USB, però, per praticità, in questo caso facciamo uso di un semplice script di Python. Infatti Python, come tantissimi altri linguaggi o ambienti, ha la possibilità di inviare e ricevere istruzioni via porta seriale. Il vantaggio pratico è quello di integrare in un unico script la funzione di controllo (in seguito vedremo che c'è anche un'altra piccola utile funzione) con quella di lettura dei dati e di registrazione dei files.

Infatti lo scopo dell'esperienza è quello di registrare il valore di una d.d.p. campionata e digitalizzata in istanti successivi. Questa registrazione avviene all'interno dello stesso microcontroller, che per questo sfrutta una piccolissima sezione di memoria (2 kB, di tipo SRAM). Se i dati rimanessero in questa memoria non sapremmo cosa farcene: l'esperienza prevede infatti di analizzarli, cioè di farci grafici, istogrammi, etc., e magari di trarre qualche conclusione fisica. È quindi necessario che essi vengano trasferiti dalla scheda di Arduino al computer. Questo trasferimento dati, che dà luogo alla scrittura di files sul disco rigido del computer, avviene appunto grazie allo script di Python.

Il diagramma logico che sta alla base dell'impiego di Arduino nelle nostre esperienze è quello mostrato molto schematicamente in Fig. 1:

1. si scrive uno sketch nell'ambiente Arduino (IDE, o Programming) che contiene, in un formato opportuno e con una sintassi simil-C, le istruzioni da impartire al microcontroller (naturalmente questo sketch lo troverete già presente nei computer di laboratorio, ma potrete eventualmente modificarlo e migliorarlo, cambiandogli nome per evitare casini);

2. lo sketch viene trasferito (upload) a Arduino tramite comunicazione seriale USB e caricato nella memoria del microcontroller (una sola volta);

3. si scrive uno script di Python che serve per controllare la partenza delle operazioni compiute da Arduino e per permettere il trasferimento e la lettura dei dati acquisiti (anche in questo caso lo script lo troverete già nei computer, e siete liberi di migliorarlo, cambiandogli nome);

4. si lancia lo script di Python, naturalmente dopo aver collegato in modo corretto i componenti necessari per l'esperienza, si aspetta un po' e, quando segnalato dalla console di Python, si sa che le misure sono state acquisite e trasferite dalla memoria di Arduino in un file registrato nel computer;

5. il file è allora pronto per essere riaperto con un altro script di Python *da fare ex-novo*, che permetterà l'analisi del campione di dati (grafici, fit, etc.); tutta l'operazione di misura può essere eseguita più volte, per esempio per verificare la presenza di fluttuazioni o per analizzare il comportamento di diverse configurazioni sperimentali, come discusso nel seguito.

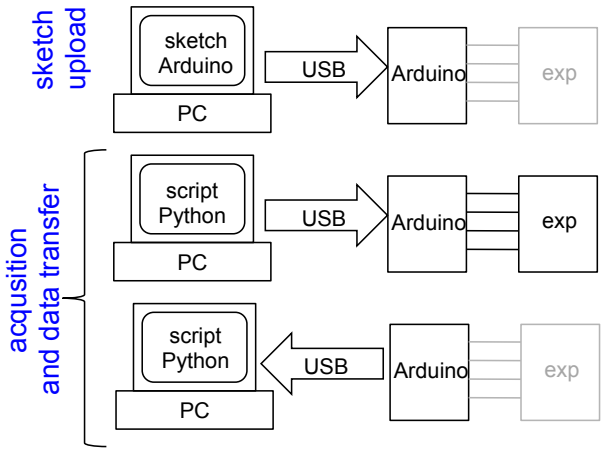

Figura 1. Diagramma logico di massima di una tipica esperienza con uso di Arduino.

A. Bit, byte, parole e caratteri

Questa breve sezione è dedicata a chiarire un po' di nomenclatura di cui faremo brevemente uso nei commenti allo sketch e allo script. L'argomento di riferimento, quello delle grandezze numeriche nella programmazione, è molto complesso e non sempre (quasi mai) ben definito, per cui sicuramente non è questa la sede per affrontarlo compiutamente. Ci limiteremo a dare poche informazioni strettamente utili, anche se talvolta incomplete e

non del tutto corrette, per capire qualche dettaglio delle istruzioni usate.

Tutti sappiamo che un *bit* è un numero binario che può assumere i valori 0 o 1. Un *byte*, indicato spesso con il simbolo B, è una sequenza di 8 bits: poiché $2^8 = 256$, un byte permette di specificare un valore intero compreso tra 0 e 255. Per ragioni storiche, qualche volta si dà il nome di *parola digitale* a una sequenza di (almeno) 2 bytes. Altrettanto per ragioni storiche, a un singolo byte si dà spesso il nome di *carattere*: il motivo è che esiste tuttora una vecchia convenzione, che risale ai tempi delle telescriventi (servivano per i telex, ovvero telegrammi), in cui un gruppo di 256 caratteri, comprendenti lettere dell'alfabeto esteso, numeri da 0 a 9, caratteri speciali e marzianetti vari, era codificato appunto in valori numerici interi compresi tra 0 e 255. Il codice di riferimento si chiama *codice ASCII* e lo potete facilmente trovare in rete.

Vale la pena di sottolineare che, in tempi di globalizzazione, fare riferimento a un codice in grado di identificare i soli caratteri alfabetici latini, e poco più, si è reso presto insufficiente. Sono quindi state sviluppate ulteriori codifiche, tra cui quella denominata *unicode*, che inizialmente si basava su parole di 2 byte, ovvero 16 bits, e che ora è estesa, potenzialmente, su ben 21 bits. Questa precisazione è funzionale ai nostri scopi considerando che la versione attuale di Python (la 3.x) tratta caratteri unicode, mentre la precedente (la 2.x, che è anche quella installata nei computer di laboratorio), era basata su caratteri ASCII. Questa circostanza comporta dei dettagli di sintassi che saranno evidenziati nel seguito (in ogni caso le istruzioni relative di Python 3.x dovrebbero essere compatibili con tutte le versioni in uso comune di Python 2.x, e viceversa).

La comunicazione seriale usata per parlare con Arduino sfrutta, normalmente, proprio dei bytes, nel senso che bytes vengono inviati e ricevuti tramite porta seriale USB. La natura seriale di questo protocollo di comunicazione implica che i bytes, ovvero i caratteri, vengano inviati e ricevuti *uno alla volta*. In altre parole, non è possibile inviare via porta seriale un valore numerico arbitrariamente grande, o una parola composta di un numero qualsiasi di caratteri, in un colpo solo. Questo aspetto non è rilevante per l'upload dello sketch, che è gestito da Arduino IDE, ma può essere importante quando si progetta lo script di Python.

III. CONFIGURAZIONE DI MISURA

Come già più volte affermato, l'esperienza prevede di eseguire in maniera automatica molte misure (centinaia o migliaia) della stessa grandezza, che, nella pratica, è una d.d.p., qui chiamata ΔV. Tale grandezza sarà *digitalizzata* da Arduino: dunque, a meno di non eseguire una *calibrazione*, della quale ci occuperemo in seguito, essa sarà data da un *numero intero* (misurato in *unità arbitrarie di digitalizzazione*, che qui chiamiamo anche *digit*) necessariamente compreso tra 0 e 1023, vista la dinamica, o profondità di digitalizzazione, di Arduino (10 bit). Il campione di misure viene temporaneamente registrato nella memoria dati (SRAM) di Arduino e di qui, al termine dell'acquisizione, trasferito al computer via porta seriale (USB).

Le tante misure vengono acquisite in successione, dunque a istanti diversi. In questa esperienza *non interessa* conoscere l'istante in cui avviene l'acquisizione, poiché non abbiamo necessità di studiare l'andamento temporale della grandezza considerata. Di conseguenza, il *time stamp*, cioè l'indicazione di quando una certa misura viene eseguita, non viene acquisito [3]. In linea di massima, gli istanti di digitalizzazione delle singole misure potrebbero essere scelti arbitrariamente, però, come sarà evidente nel seguito, è estremamente più semplice impostare l'esperimento in modo che essi siano *pressoché* equispaziati.

Facciamo subito due osservazioni molto importanti, anche se non necessariamente rilevanti per la presente esperienza:

1. gli intervalli di tempo tra una misura e la successiva sono gestiti, cioè, di fatto, decisi, dal programma che gira nel microcontroller. Di conseguenza essi sono affetti da un'incertezza che, in generale, può essere non trascurabile [4];

2. il campionamento avviene in un breve intervallo di tempo che è "triggerato" (avviato) dalle istruzioni del programma che gira nel microcontroller, ma che normalmente ha luogo in una frazione, non completamente nota, del tempo che intercorre fra due digitalizzazioni, o campionamenti, successivi [5].

A. Partitore con potenziometro

Poiché potrebbe essere di interesse (e lo è, di fatto) misurare d.d.p. di diverso valore, e tenendo conto che in laboratorio non è disponibile un generatore di tensione variabile, è ovvio che la d.d.p. da misurare venga prodotta da un *partitore di tensione*. Per evitare di essere vincolati a valori prefissati del rapporto di partizione, come si verifica quando si ha a disposizione un numero limitato di resistori, per questa esperienza il partitore di tensione include un resistore variabile, o *potenziometro*.

Il potenziometro è un dispositivo elettro-meccanico che, almeno grossolanamente, può essere visto come un contatto strisciante mobile su una pista di materiale conduttore (ad alta resistività). Il contatto strisciante è solidale a un alberino, la cui rotazione, quindi, fa assumere una diversa resistenza tra contatto strisciante stesso e estremità della pista conduttiva. La Fig. 2(a) illustra schematicamente la realizzazione e riporta un simbolo convenzionale del potenziometro. Notate che, in genere, il potenziometro ha tre terminali, come un figura. La resistenza tra contatto strisciante (terminale "centrale") e uno dei due estremi della pista conduttiva (uno degli altri

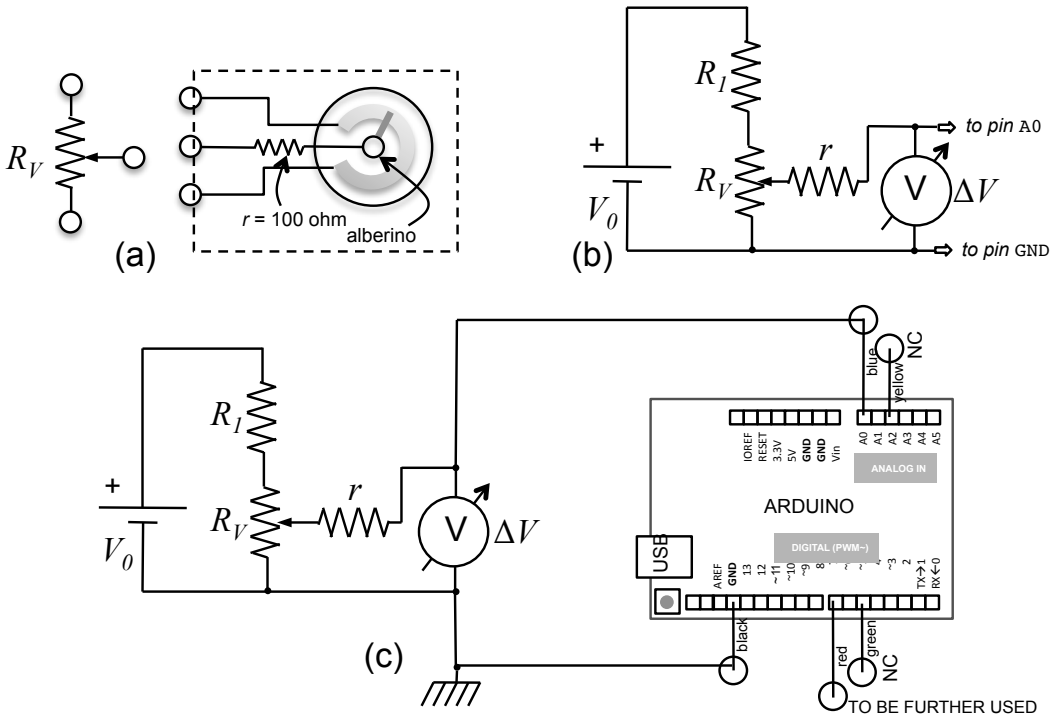

Figura 2. Rappresentazione schematica e costruttiva del potenziometro "visto da sotto" (a), schema del partitore di tensione (b), configurazione del circuito di misura comprendente Arduino (c). Nel pannello (a) è riportato uno dei simboli con cui si indica un potenziometro negli schemi elettronici. Nel pannello (c) è rappresentata una visione molto schematica e non in scala della scheda Arduino Uno rev. 3 SMD edition usata nell'esperienza. Ci sono cinque collegamenti a altrettanti pin della scheda che terminano con boccole volanti di diverso colore, secondo quanto indicato in figura: solo due boccole devono essere collegate (NC significa non collegato). Notate che la boccola rossa collegata al pin `7` potrà eventualmente essere impiegata secondo quanto descritto in seguito. La "spazzolina" sulla linea di circuito che va al pin `GND` indica un collegamento a massa, ovvero a terra.

due terminali) varia tra 0 (circa) e un valore massimo R_V in funzione della rotazione dell'alberino; nel contempo, la resistenza tra contatto strisciante e l'altro terminale varia tra R_V e (circa) 0. Per la maggior parte dei potenziometri disponibili in laboratorio (non tutti) $R_V = 4.7$ kohm.

La schema del partitore di tensione è rappresentato in Fig. 2(b), dove il generatore di d.d.p. è quello disponibile in laboratorio ($V_0 \simeq 5$ V). Si vede come siano presenti altre due resistenze: R_1, da scegliere nel banco delle resistenze (nelle mie prove $R_1 = 680$ ohm, nominali) e $r = 100$ ohm (nominali), saldata direttamente al terminale centrale del potenziometro, dunque parte del telaietto che ospita questo dispositivo. Queste resistenze sono incluse nel circuito a fini "protettivi", cioè per evitare che nel partitore fluisca una corrente troppo alta (comporterebbe possibile bruciatura del fusibile, e anche del potenziometro, che può dissipare una potenza massima di 1 W, tipicamente). Per come è configurato il circuito, la rotazione dell'alberino del potenziometro permette di ottenere in uscita dal partitore una d.d.p. variabile con continuità da (circa) 0 a un certo valore massimo, determinato dai valori di V_0, R_1, r, R_V (potete facilmente dimostrarlo con le regoline dei partitori di tensione) [6]. Questa d.d.p. può, e deve, essere misurata continuativamente: allo scopo si usa il tester digitale, che ha resistenza interna sicuramente maggiore di quella del potenziometro. Nel mio esempio, ho ottenuto $\Delta V \sim 0 - 4.5$ V.

L'uscita del partitore deve essere inviata all'ingresso (porta analogica) di Arduino prescelto per la misura, che in questo esempio corrisponde al pin `A0`. Il pin in questione si trova, assieme ad altri, su un connettore a pettine di tipo femmina. Su di esso è innestato un maschio con dei cavetti saldati che terminano con boccole volanti: si usano colori diversi per cavetti e boccole diverse, e quello del pin `A0` è il blu. Ricordate che la misura di una tensione richiede di usare due fili (è una *differenza* di potenziale): l'altro filo, che deve essere collegato alla linea connessa con il negativo dell'alimentatore, va a uno dei pin marcati con `GND` (ground, cioè terra, ovvero massa). Boccola e filo del collegamento di massa, o terra, sono di

colore nero. Ricordate anche che, per come è costruito, il digitalizzatore di Arduino accetta in ingresso solo d.d.p. *positive* (o nulle) rispetto alla linea di terra. Pertanto fate la massima attenzione a rispettare le polarità: la boccola di uscita del generatore di d.d.p. che deve essere collegata alla linea di terra (boccola volante nera collegata al pin `GND` di Arduino) è quella *nera*. Se sbagliate, Arduino può salutarvi.

I connettori a pettine di cui sono dotate le schede Arduino in uso in laboratorio hanno anche altre connessioni: in linea di massima non dovete usare altre boccole, tranne eventualmente quella rossa, collegata alla porta digitale corrispondente al pin `7`, da usare per gli scopi di calibrazione che descriveremo in seguito. Le connessioni da effettuare sono schematizzate in Fig. 2(c).

IV. LO SCRIPT DI PYTHON

Anche se la logica suggerirebbe di partire dalle istruzioni dello sketch di Arduino, iniziamo con il commento allo script di Python.

Lo script richiede di importare due pacchetti che finora non abbiamo mai usato: il pacchetto `serial`, che serve per gestire (al meglio) la comunicazione seriale USB, e il pacchetto `time`, che permette di eseguire dei cicli di attesa con un tempo controllato. Nello script compaiono infatti delle istruzioni del tipo `time.sleep(2)` che producono un'attesa di 2 s (il valore può essere ovviamente modificato, l'unità di misura è secondi), precauzionalmente necessaria per evitare di avere problemi di intasamento della comunicazione seriale.

Come già affermato, il numero di misure distinte che possono essere eseguite e registrate nella memoria di Arduino in una singola acquisizione è limitato (sono 600 nell'esempio qui considerato). Dato che potrebbe essere utile creare un campione di misure più grande, lo script è predisposto per eseguire un loop di diverse acquisizioni, permettendone la registrazione su un unico file [7]. Questo loop è avviato dall'istruzione `for j in range (1,nacqs+1):`, con `nacqs` da definire nello script. State attenti alla particolare sintassi: in Python l'istruzione del loop termina con un "`:`" e le istruzioni che devono essere eseguite nel ciclo sono *indentate* (tabulate, per usare il linguaggio delle macchine da scrivere), cioè rientrate rispetto al margine sinistro dello script. Inoltre nella parte iniziale dello script si stabilisce la directory che conterrà i dati. Di default, la directory dove sono raccolti i dati nei computer di laboratorio è `../dati_arduino/`. È anche necessario fornire il nome del file, che dovrete stabilire secondo i vostri gusti.

Dopo aver inizializzato la porta seriale, cioè attribuito alla variabile `ard` un valore identificativo della porta USB a cui la scheda Arduino è collegata (la sintassi è peculiare e dipendente dal sistema operativo) e specificato che la comunicazione avverrà alla velocità di 9600 Baud (nella comunicazione usata, un Baud dovrebbe corrispondere a 1 B/s), non molto elevata ma sicuramente adeguata agli scopi, lo script scrive sulla porta seriale un determinato valore. L'istruzione corrispondente è, per esempio, `ard.write(b'1')` che significa che alla porta seriale corrispondente alla variabile `ard` (sarebbe il nostro Arduino) viene inviato in scrittura (`.write`, sintassi in cui si riconosce bene la concatenazione attraverso il punto in uso con Python) un carattere di tipo ASCII (il `b` dell'istruzione, che a rigore non serve usando Python 2.x) costituito dal carattere '1'.

Come sarà discusso in seguito, l'invio di questo carattere ha la duplice funzione di far partire l'acquisizione da parte di Arduino e di indicargli quanto deve valere l'intervallo temporale *nominale* Δt tra una presa dati e la successiva. L'unità di misura è, in questa esperienza, 10 ms, per cui l'"1" significa che i dati saranno campionati con intervalli di *circa* 1×10 ms $= 10$ ms. La scelta di questo parametro non influenza i risultati della presente esperienza, mentre invece sarà critica per altre esperienze da svolgere in futuro.

Quindi lo script attende finché sulla porta seriale, continuamente monitorata, non compaiono dei dati. Arduino li rende disponibili al termine dell'acquisizione, dunque in questo modo ci si garantisce che i dati vengano trasferiti al computer quando sono effettivamente pronti. Poiché la comunicazione seriale prevede lo scambio di dati uno alla volta, la porta seriale viene letta all'interno di un loop, con un indice che gira fino al numero di dati acquisiti, cioè delle misure fatte (600, nell'esempio considerato). Notate la sintassi abbastanza specifica per l'operazione di lettura di Arduino: essa prevede di leggere un dato alla volta attraverso l'istruzione `data = ard.readline().decode()`, che contiene anche l'istruzione di decodifica dei dati stessi, che devono essere interpretati come numeri (interi). Ogni dato viene aggiunto al file (di testo) dei dati. Al termine di ogni acquisizione del ciclo viene chiusa la comunicazione seriale con Arduino attraverso l'istruzione `ard.close()` e al termine delle operazioni il file dei dati viene anche chiuso con l'istruzione `outputFile.close()`.

Nel corso di tutto il processo è prevista la scrittura sulla console (cioè sul terminale) di indicazioni di progresso: alla fine di tutto compare un bell'`End`.

Lo script, debitamente commentato, è riportato qui di seguito; esso si trova in rete sotto il nome di `ardu_multicount_v1.py`.

```
import serial # libreria per gestione porta seriale (USB)
import time   # libreria per temporizzazione
nacqs = 1 # numero di acquisizioni da registrare (ognuna da 600 punti)
```

```
Directory='../dati_arduino/' # nome directory dove salvare i file dati
FileName=(Directory+'dataXX.txt') # nomina il file dati <<<< DA CAMBIARE SECONDO GUSTO
outputFile = open(FileName, "w+" ) # apre file dati in scrittura
for j in range (1,nacqs+1): # avvia il ciclo di acquisizioni multiple (una sola se nacqs = 1)
        print('Apertura della porta seriale\n') # scrive sulla console (terminale)
        ard=serial.Serial('/dev/ttyACM0',9600)  # apre la porta seriale (da controllare come viene
denominata, in genere /dev/ttyACM0)
        time.sleep(2)   # aspetta due secondi per evitare casini
        ard.write(b'1')#intervallo (ritardo) tra le acqusizioni in unita' di 10 ms <<<<
questo si puo' cambiare (default messo a 10 ms)
        print('Start Acquisition ',j, ' of ',nacqs) # scrive sulla console (terminale)
        # loop lettura dati da seriale (sono 600 righe, eventualmente da aggiustare)
        for i in range (0,600):
                data = ard.readline().decode() # legge il dato e lo decodifica
                if data:
                        outputFile.write(data) # scrive i dati nel file
        ard.close() # chiude la comunicazione seriale con Arduino
        print('Acquisition ',j,' completed\n') # scrive sulla console (terminale)
outputFile.close() # chiude il file dei dati
print('End') # scrive sulla console che ha finito
```

V. LO SKETCH DI ARDUINO

Lo sketch di Arduino è scritto in un linguaggio che somiglia molto al C. In questo linguaggio si fa uso molto spesso di pseudo-funzioni, cioè gruppi di istruzioni che non ritornano un valore numerico. A queste pseudo-funzioni si fa riferimento con l'istruzione `void { }` (le parentesi graffe comprendono le istruzioni associate alla pseudo-funzione). Notate che pressoché tutte le singole istruzioni contenute tra le parentesi graffe devono necessariamente *terminare con un punto e virgola* (fanno eccezione, per esempio, le istruzioni che avviano un loop, per le quali il punto e virgola non deve essere usato).

Nei casi semplici, a cui fortunatamente appartiene il nostro sketch, Arduino richiede che lo sketch stesso sia suddiviso in diverse parti. Il nostro sketch è di fatto suddiviso in tre sezioni: (i) dichiarazione delle variabili; (ii) inizializzazione del microcontroller; (iii) istruzioni necessarie per le specifiche operazioni previste.

Vediamo e commentiamo brevemente il contenuto di queste tre parti.

A. Dichiarazione delle variabili

La dichiarazione delle variabili è necessaria (in C, non in Python, come sapete) affinché esse possano essere correttamente interpretate nel programma. Si possono definire delle variabili vere e proprie, oppure delle *costanti*, cioè delle grandezze che non verranno mai modificate dal programma. A seconda di quello che devono rappresentare, esse saranno identificate come variabili secche (scalari) o array (vettori), intere (segnate o meno, eventualmente "lunghe") o reali (eventualmente a doppia precisione).

A seconda della tipologia di definizione, cambia la quantità di memoria allocata per la variabile. Vista l'esiguità dello spazio di memoria disponibile nel microcontroller, è sempre consigliabile definire le variabili per quello che effettivamente serve. Per esempio, un intero standard (`int` o `unsigned int`, a seconda che debba o non debba assumere valori negativi) occupa 2 bytes, cioè due caratteri, e permette quindi di individuare $2^{8+8} = 2^{16}$ valori interi differenti; un intero `long` richiede invece 4 bytes. Se, come in questo caso, le variabili in uso sono fatte di numeri interi abbastanza "piccoli" (il risultato della misura è, di fatto, un numero intero piccolo, compreso tra 0 e 1023), conviene definirle come `int`.

Il resto di questa sezione dello sketch, che è riportata qui nel seguito, è piuttosto auto-esplicativa: notate che `analogPin_uno` e `digitalPin_uno` sono le costanti intere che indicano i pin da impiegare come porte per la lettura (analogica) e per fornire in uscita un valore di d.d.p. che sarà utile per la calibrazione, cioè le porte corrispondenti ai pin `A0` e `7` della scheda. Infine la variabile intera `start` serve come *flag*: nel seguito dello sketch ci sarà un ciclo pronto a partire quando questa variabile diventerà diversa da zero, mentre la variabile intera `delay_ms` contiene il ritardo (in ms) tra una digitalizzazione e la successiva.

Osservate che, per evitare letture artificiose che potrebbero verificarsi all'inizio del ciclo di misura, la prima lettura viene scartata [8]. A questo scopo essa viene messa nella variabile intera `trash`, che è anche dichiarata nel blocco.

```
//Dichiarazione variabili
```

```
const int analogPin_uno=0;  // Decide di usare pin A0 per lettura V1
const int digitalPin_uno=7; // Decide di usare pin 7 per uscita digitale
int i=0;  // Definisce la variabile intera che viene incrementata nel loop
int V1[600];  // Definisce l'array di interi per memorizzare V1 (d.d.p. letta da analogPin_uno)
int delay_ms;  // Definisce la variabile intera che contiene il ritardo tra due step successivi (in unita'
int start=0;  // Crea il flag di controllo (che diventa uno alla fine del processo)
int trash; // Crea la variabile intera che conterra' la prima lettura (da buttare)
```

B. Inizializzazione

Le istruzioni di inizializzazione di Arduino devono essere incluse nella pseudo-funzione chiamata `setup()`. Pertanto esse iniziano con `void setup(){}` e le istruzioni relative sono contenute tra le parentesi graffe.

L'inizializzazione richiede di aprire la porta seriale preparandola a funzionare a 9600 Baud (`Serial.begin(9600);`), di pulirne per sicurezza il buffer (`Serial.flush();`), di definire di uscita la porta indicata dalla variabile `digitalPin_uno` e di porla a livello "alto", per gli scopi di cui tratteremo in seguito. Notate che non è necessario definire la porta analogica come input, essendo questa la configurazione di default.

La parte di sketch è la seguente:

```
//Inizializzazione
void setup()
{
  pinMode(digitalPin_uno,OUTPUT); // Configura il pin digital_Pin_uno come output digitale
  digitalWrite(digitalPin_uno,HIGH); // E lo pone "alto"
  Serial.begin(9600);  // Inizializzazione della porta seriale
  Serial.flush(); // Svuota il buffer della porta seriale
}
```

C. Il loop

Le istruzioni vere e proprie del programma sono inserite in una pseudo-funzione che prevede un ciclo ed è pertanto denominata `void loop(){}`; come al solito, anche qui le istruzioni del loop sono contenute tra le parentesi graffe.

Questo ciclo inizia con un'istruzione di monitoraggio della porta seriale, necessario perché, dopo aver caricato lo sketch nel microcontroller, esso aspetta per partire di avere ricevuto via seriale (USB) la comunicazione prodotta dallo script di Python. Allo scopo provvede il comando `Serial.available()`, che restituisce un valore diverso da zero quando qualcosa si viene a trovare sulla porta seriale.

Il qualcosa in questione è il singolo carattere (byte) inviato dallo script di Python. Ricordiamo che esso contiene, in origine, un numero intero che, moltiplicato per 10, deve dare l'intervallo *approssimativo* in ms tra due istanti successivi di campionamento. Il *numero* di ms di questo intervallo è contenuto nella variabile `delays` che è costruita dalla lettura della porta seriale sfruttando un trucchettino. Infatti quello che originariamente, nelle nostre intenzioni, era un numero intero, è stato necessariamente convertito in un byte, ovvero in un carattere ASCII, prima di essere inviato attraverso porta seriale dal computer a Arduino. Per essere riconvertito in intero esso deve essere decodificato, operazione che viene effettuata con l'istruzione `Serial.read`. I numeri interi sono codificati ASCII in ordine nell'intervallo compreso tra il decimale 48, corrispondente al carattere `'0'`, e il 57, corrispondente al carattere `'9'`. Di conseguenza la decodifica, per esempio, del carattere '5' dà luogo al numero intero 53. Dunque "sottrarre lo '0"', che è codificato con il decimale 48, come fatto nello sketch, consente di ricavare il numero originario ($53 - 48 = 5$), che poi va, come detto, moltiplicato per 10 allo scopo di definire la variabile `delay_ms`.

Di seguito, dopo aver svuotato il buffer della porta seriale, la variabile `start` viene posta a 1 e l'acquisizione comincia. Per scopi puramente precauzionali, e probabilmente inutili, prima di iniziare le misure viene imposta ad Arduino una pausa di 0.5 s attraverso l'istruzione `delay(500)` (l'unità di misura è ms). Questa pausa tranquillizza nei confronti di possibili intasamenti nel funzionamento del microcontroller.

A questo punto inizia il ciclo di misure. La digitalizzazione del segnale sulla porta di ingresso indicata dalla variabile `analogPin_uno` avviene attraverso l'istruzione `analogRead(analogPin_uno);`, che restituisce un intero compreso tra 0 e 1023. Come già affermato, la prima misura viene scartata, e dunque messa nella variabile `trash` che poi viene "cestinata" (non considerata). Quindi ha inizio il ciclo di misure vere e proprie, che, in questo esem-

pio, si svolge su 600 digitalizzazioni. L'istruzione che avvia il ciclo è `for(i=0;i<600;i++)`, con ovvia sintassi; le istruzioni del ciclo sono comprese tra parentesi graffe. Il risultato delle misure va nell'array `V1[i]` grazie all'istruzione `V1[i] = analogRead(analogPin_uno);`. Questa istruzione è seguita dalla `delay(delay_ms);`, che temporizza il campionamento su intervalli distanti temporalmente per il valore impostato [9]. Ricordiamo ancora che la temporizzazione effettiva potrebbe essere diversa da quella desiderata a causa delle latenze e dei cicli di attesa del microcontroller, per cui l'impostazione va presa come approssimativa [10].

Terminata l'acquisizione dei 600 punti sperimentali comincia il ciclo di scrittura dei dati sulla porta seriale. Il modo con cui essi sono scritti non è molto elegante, ma garantisce di avere dati che possono facilmente essere letti da Python. I dati sono organizzati in righe al termine delle quali si "va a capo" per iniziare una nuova riga. Questo è realizzato dall'istruzione `Serial.println(V1[i]);`.

Alla fine di questo ciclo di scrittura la variabile flag viene annullata, in modo da uscire dall'acquisizione, e la porta seriale viene svuotata.

La corrispondente sezione di sketch è riportata nel seguito (l'intero sketch si trova in rete sotto il nome `ardu.ino`):

```
//Ciclo di istruzioni del programma
void loop()
{
    if (Serial.available() >0) // Controlla se il buffer seriale ha qualcosa
      {
        delay_ms = (Serial.read()-'0')*10; // Legge il byte e lo interpreta come ritardo (unita' 10 ms)
        Serial.flush(); // Svuota il buffer della seriale
start=1; // Pone il flag start a uno in modo da avviare l'acquisizione
      }
  if(!start) return // Quando il flag è zero termina l'acquisizione
    delay(500); // Attende 0.5s
    trash=analogRead(analogPin_uno); // Fa una prima lettura (che non verra' considerata) per evitare spike
    delay(delay_ms); // Aspetta il tempo impostato
    for(i=0;i<600;i++)  // Avvia il loop di acquisizione
      {
       V1[i]=analogRead(analogPin_uno);   // Legge il pin analogPin_uno
       delay(delay_ms);   // Aspetta il tempo impostato
      }
     for(i=0;i<600;i++)  // Nuovo ciclo che scorre l'array di dati e lo scrive sulla seriale
     {
       Serial.println(V1[i]);  // Scrive il dato sulla seriale e va a capo
     }
    start=0; // Annulla il flag in modo da uscire dall'acquisizione
    Serial.flush(); // Svuota il buffer della porta seriale
  }
```

VI. ANALISI DI DATI ESEMPIO

L'esperimento descritto consente, in poche parole, di ottenere un campione di misure di una grandezza, la d.d.p. presente tra pin `A0` e `GND`, supposta costante. Anche se l'analisi non è particolarmente significativa dal punto di vista della fisica coinvolta, questo campione può essere analizzato. Oltre ad avere ovvie motivazioni didattiche, l'analisi può servire per evidenziare alcuni aspetti di base nella digitalizzazione e campionamento automatizzati. Per esempio, possiamo porci l'obiettivo di

- osservare la distribuzione del campione e calcolarne la deviazione standard sperimentale. Qui lo scopo potrebbe essere quello di verificare se l'incertezza che abbiamo deciso di attribuire alle misure di Arduino (± 1 bit, ovvero ± 1 digit, nel nostro linguaggio) è ragionevole;
- verificare sul campo cosa significa calibrare uno strumento di misura, quale per noi, in questa esperienza, è Arduino.

Nel seguito, si riportano alcuni risultati delle analisi condotte su campioni esempio, insieme a qualche commento generale.

A. Misura di una d.d.p. "costante"

In primo luogo è stato acquisito un campione costituito da 3000 misure (cioè costruito scegliendo `nacqs = 5` nello script `ardu_multicount_v1.py`) della d.d.p. in uscita al partitore per una certa scelta della posizione dell'alberino del potenziometro, ovviamente tenuta fissa durante l'acquisizione stessa. Nel mio esempio, a tale scelta corrispondeva $\Delta V = (2.65 \pm 0.02)$ V, misurata con il tester digitale. Questa d.d.p. rimaneva costante (entro l'errore) al distacco della porta di ingresso di Arduino, per cui per questa misura la resistenza di ingresso di Arduino è stata considerata sufficientemente alta da produrre effetti trascurabili.

Il primo interesse è, naturalmente, quello di visualizzare il campione. Allo scopo è stato costruito uno script di Python che legge il file di dati e lo grafica per punti. Secondo la nostra convenzione, ai dati sperimentali, espressi in unità arbitrarie di digitalizzazione ovvero, nel nostro linguaggio, digit, è attribuita un'incertezza pari a ± 1 nelle unità specificate. La Fig. 3 mostra il risultato nel pannello superiore: si vede chiaramente che la lettura è pressoché costante.

Lo script di Python provvede anche a determinare il valore medio (indicato come `avg` e riportato con un segmento punto-linea nel grafico) e la deviazione standard, che per questo esempio risultano rispettivamente $\overline{y} = 536.31$ digit e $\sigma_y = 0.49$ digit. Inoltre, come succede spesso quando si analizza un campione di dati, può essere utile graficare l'*istogramma delle occorrenze*, come mostrato nel pannello in basso della stessa figura.

Per realizzare l'istogramma con Python esistono diverse possibilità. La più semplice consiste nell'uso dell'istruzione `pylab.hist`, che provvede a costruire e visualizzare l'array di istogramma delle occorrenze. Si ricorda che questa istruzione contiene un certo numero di argomenti: `pylab.hist(sample, bins = xx, range = (*,*), histtype = ??, color = ??,normed='')` , dove `sample` è l'array che si intende analizzare (si chiama `y` nel nostro esempio, e ovviamente contiene i dati del campione) e il resto è sufficientemente auto-esplicativo. Per avere una corretta rappresentazione, occorre aggiustare il range e il numero di bin in modo tale che *i vari bin corrispondano a numeri interi*, che sono quelli effettivamente misurati. Inoltre, essendo interessati all'istogramma delle occorrenze, *non* è necessario, e anzi è sconsigliato, arrangiarsi per ottenere l'istogramma delle frequenze (normalizzate). Per vostra comodità, in rete è disponibile sotto il nome `ardu_hist.py` uno script che calcola $\overline{y}$ e σ_y, e produce grafico e istogramma come in Fig. 3, ma siete fortemente invitati a provvedere da soli alla redazione dello script necessario per l'analisi del campione.

Il campione considerato non è particolarmente interessante. La distribuzione ottenuta dimostra infatti che la dispersione delle misure è inferiore a un digit, confermando come valida l'ipotesi di porre la barra di errore delle misure pari a ± 1 digit. La residua dispersione rispetto al valore medio è dovuta a fluttuazioni attribuibili sia alla misura (il carattere stocastico del conteggio tipico del digitalizzatore) che al circuito sotto esame (variazioni della d.d.p. in uscita dal partitore, dovute a fluttuazioni del funzionamento del circuito e della d.d.p. erogata dal generatore). Ci sono poi anche altre possibili origini ("esterne") per tali fluttuazioni, su cui torneremo in seguito.

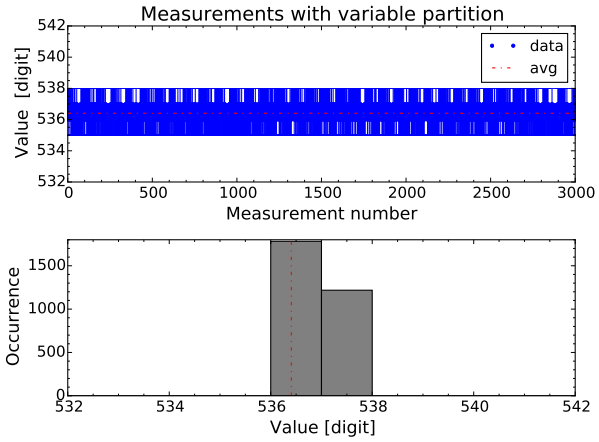

Figura 3. Esempio di analisi del campione di dati acquisito per $\Delta V = (2.65 \pm 0.02)$ V, misurata con il tester digitale. Il pannello superiore riporta il grafico delle misure, a cui è stata attribuita un'incertezza ± 1 digit di origine stocastica, il pannello inferiore riporta l'istogramma delle occorrenze. Valore medio e deviazione standard sperimentale del campione considerato sono rispettivamente $\overline{y} = 536.31$ digit e $\sigma_y = 0.49$ digit e vengono indicate con segmenti punto-linea nei grafici.

Dal punto di vista fisico, viste le dimensioni del campione e l'origine prevalentemente stocastica delle fluttuazioni, avremmo potuto aspettarci una distribuzione "strutturata", per esempio una distribuzione normale con andamento Gaussiano (centrata su $\overline{y}$ e di larghezza proporzionale a σ_y). Nella maggior parte dei casi, invece, la distribuzione non rivela alcuna "forma", che d'altronde è in accordo con il fatto che la deviazione standard è minore del singolo bin dell'istogramma. In altri termini, la sensibilità della nostra misura non è sufficiente per determinare la distribuzione. Nella Sez. VIII mostreremo delle distribuzioni ottenute facendo lavorare Arduino a una sensibilità maggiore.

VII. CALIBRAZIONE

Per gli scopi della nostra analisi, cioè per lo studio della distribuzione del campione, è naturalmente del tutto sufficiente rappresentare le misure nella scala arbitraria delle unità di digitalizzazione. Quindi questa è una di quelle situazioni sperimentali nelle quali non è necessario esprimere la misura in unità "fisiche". L'analisi svolta, inoltre, dimostra che è ragionevole imporre alle misure digitalizzate con Arduino una barra di errore unitaria (la devia-

zione standard sperimentale è ben compresa nella barra di errore e quindi così facendo si compie una piccola, ma ragionevole, sovrastima dell'incertezza).

Tuttavia in alcuni casi è necessario rapportare il risultato della digitalizzazione a unità fisiche, cioè, in questo caso, Volt. Per questo è necessaria una *calibrazione* dello strumento, cioè del digitalizzatore di Arduino.

In prima approssimazione, questa calibrazione può essere compiuta considerando la lettura [$\Delta V = (2.65 \pm 0.02)$ V per l'esempio di Fig. 3] fatta con il tester digitale nelle stesse condizioni dell'acquisizione del campione. Supponendo, secondo le aspettative, che la digitalizzazione sia un processo lineare, il fattore di calibrazione risulta per questo esempio $\xi = \Delta V/\overline{y} = (4.94 \pm 0.03)$ mV/digit, dove abbiamo debitamente indicato l'*incertezza di calibrazione*, che è dominata dall'incertezza con la quale riusciamo a misurare ΔV con il tester digitale. Dunque abbiamo eseguito una calibrazione "per confronto", in cui abbiamo comparato il risultato di una *singola* misura del nostro strumento (Arduino) con quella di uno strumento di riferimento (il tester digitale), dotato a sua volta di un'incertezza nota e tutt'altro che trascurabile.

Come già affermato, questo fattore di calibrazione, che qui supponiamo costante nel tempo, dipende dal valore di una tensione di riferimento V_{ref} prodotta internamente da Arduino a partire dalla sua tensione di alimentazione. Nelle nostre esperienze, Arduino è alimentato direttamente tramite la porta USB del computer a cui è collegato, che fornisce tipicamente una d.d.p. attorno a 5 V [11]. È ragionevole che essa sia variabile da computer a computer, o da scheda a scheda, e anche che ci siano delle fluttuazioni che, nella nostra stima dell'incertezza di calibrazione, abbiamo trascurato. Di default, dunque, Arduino costruisce V_{ref} in modo che essa sia "simile" alla tensione di alimentazione, cioè sia attorno a 5 V.

Secondo le specifiche di Arduino, il valore di V_{ref} di default dovrebbe anche corrispondere, almeno in prima approssimazione, con la massima d.d.p. che può essere erogata dalle porte digitali di Arduino. Come ricordate, nel nostro sketch abbiamo attivato una di queste porte, quella corrispondete al pin 7, e l'abbiamo posta a livello alto. Questo significa che, in prima approssimazione, essa dovrebbe trovarsi a una d.d.p. (sempre rispetto alla massa, o terra) $V_{7,high} \simeq V_{ref}$. Poiché $V_{7,high} \simeq V_{ref}$ è anche la massima lettura che può essere eseguita da Arduino, quella corrispondete al livello 1023 della digitalizzazione, il fattore di calibrazione dovrebbe poter essere determinato anche dal rapporto $\xi' \simeq V_{7,high}/1023$. Nel mio esempio, ho letto $V_{7,high} = (5.03 \pm 0.03)$ V, da cui $\xi' = (4.92 \pm 0.03)$ mV/digit, che è effettivamente in accordo con ξ determinato sopra.

A. Linearità, offset e calibrazione con best-fit

Questa sezione va nello specifico presentando un metodo di calibrazione un po' più raffinato, ma che *non è obbligatorio* eseguire nell'esperienza pratica perché richiede un po' troppo tempo.

È evidente che le calibrazioni che abbiamo determinato sopra sono eseguite per (due distinti) singoli punti sperimentali. Se vogliamo estendere la calibrazione all'intero range di valori misurabili, dobbiamo innanzitutto essere certi che la relazione che lega ΔV con $\overline{y}$ sia *sempre* lineare. Inoltre dobbiamo anche garantirci che, in questa relazione, non ci siano termini costanti, ovvero di *offset*.

Tutto questo può essere facilmente verificato eseguendo più misure $\overline{y}_j$ corrispondenti a più valori ΔV_j, facilmente ottenibili ruotando l'alberino del potenziometro, e facendo un best-fit lineare delle coppie di dati così ottenuti.

La Fig. 4 mostra (pannello superiore) il risultato della procedura da me ottenuto (con poche misure). Le barre di errore (che ci sono, anche se poco visibili) sono state determinate dall'incertezza di lettura del tester per ΔV_j e dalla deviazione standard sperimentale per $\overline{y}_j$, sempre ottenuta su campioni di 3000 misure. Nel best-fit, che ho eseguito analiticamente, ho tenuto conto tramite propagazione dell'errore ("in quadratura") di entrambi le incertezze. I risultati sono

$$a = (15.9 \pm 5.6) \text{ mV} \tag{1}$$

$$b = (4.895 \pm 0.014) \text{ mV/digit} \tag{2}$$

$$\chi^2/\text{ndof} = 0.4/8 \, , \tag{3}$$

con a e b parametri che rappresentano rispettivamente termine costante (intercetta) e pendenza della retta di fit, entrambi opportunamente dimensionati. Il basso valore del χ^2_{rid}, probabilmente legato alla sovrastima delle incertezze su ΔV_j, e soprattutto il grafico dei residui normalizzati, che, come mostra il pannello inferiore di Fig. 4, non presenta evidenti "tendenze" rispetto al valore medio (mostrato con un segmento punto-linea), suggeriscono che l'andamento lineare sia ben soddisfatto. Tuttavia il coefficiente a (intercetta) ben diverso da zero mostra che nel digitalizzatore sono presenti degli offset non trascurabili [12]; di conseguenza il fattore di calibrazione è diverso rispetto a quello determinato da singole misure, e prima riportato. Naturalmente tale differenza non ha particolare rilevanza pratica per i nostri scopi e quindi, anche in futuro, quando saremo eventualmente chiamati a determinare il fattore di calibrazione potremo usare quello ottenuto dalla singola misura, se non richiesto diversamente.

VIII. MISURE "A MAGGIORE SENSIBILITÀ"

Fermo restando che il numero di livelli disponibili per la digitalizzazione è sempre di 1024, cioè che il valore della misura è sempre compreso tra 0 e 1023, è evidente che per aumentare la sensibilità dello strumento (Arduino) si può agire sul valore di V_{ref}. Infatti, almeno in prima approssimazione, la più piccola differenza di potenziale apprezzabile, cioè quella che corrisponde al bit unitario, è pari a $V_{ref}/1023$.

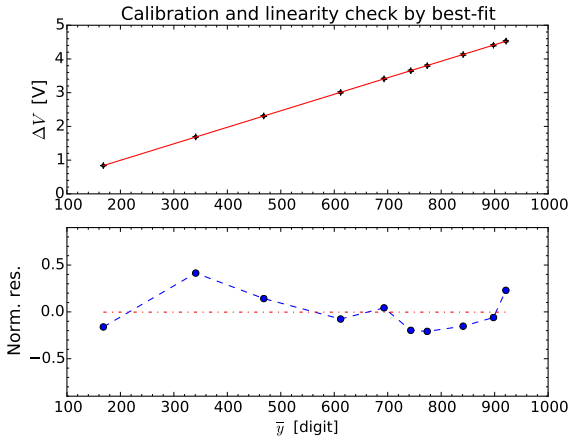

Figura 4. Esempio di calibrazione e controllo della linearità di Arduino eseguito tramite best-fit, secondo quanto discusso nel testo. Il pannello superiore riporta i dati sperimentali e il best-fit (analitico) secondo una retta non passante per l'origine (linea continua). Il pannello inferiore mostra i residui normalizzati assieme al loro valore medio (segmento punto-linea).

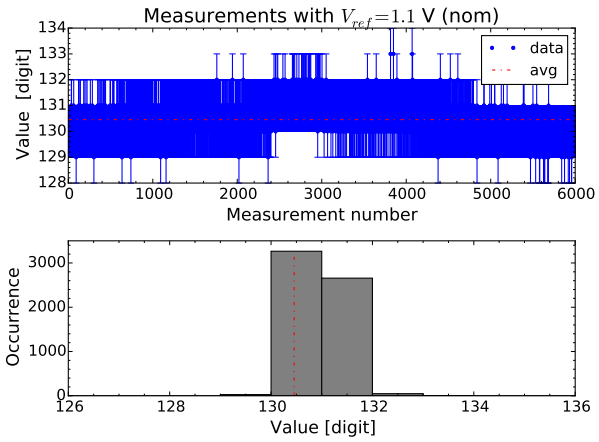

Figura 5. Analogo di Fig.3 per un campione di 6000 misure acquisito impostando $V_{ref} = 1.1$ V nominali in Arduino. Per l'acquisizione considerata $\Delta V = (139.1 \pm 0.7)$ mV, misurata con il tester digitale, $\overline{y} = 130.26$ digit e $\sigma_y = 0.53$ digit.

Oltre al default di cui abbiamo già fatto uso, Arduino consente diverse scelte per V_{ref}. Per esempio, essa può essere fornita dall'esterno usando un apposito pin della scheda. Questa scelta, però, richiede di produrre esternamente un riferimento stabile di tensione, obiettivo al di fuori delle nostre possibilità attuali. Fortunatamente Arduino prevede un'altra opzione, consistente nello scegliere un riferimento *interno* che ha valore *nominale* $V_{ref} = 1.1$ V, dunque circa cinque volte minore di quello di default [13]. Questo riferimento viene attivato da una semplice istruzione da mettere nello sketch (nel blocco di inizializzazione): `analogReference(INTERNAL);`. Uno sketch con questa istruzione è disponibile nei computer di laboratorio sotto il nome `ardu1V1.ino`.

Naturalmente quando viene impostata questa scelta per V_{ref} occorre garantirsi, pena possibili danneggiamenti, che la d.d.p. da misurare *sia minore di* 1.1 V. Grazie all'uso del partitore con il potenziometro, che consente di scegliere il valore di ΔV in modo continuo, questa richiesta può essere facilmente soddisfatta. La Fig. 5 mostra un esempio di campione così acquisito: si vede che la distribuzione tende ad estendersi su diversi bin, anche se nulla si può dedurre sulla sua forma. Per referenza, nell'esempio considerato (campione di 6000 misure) si ottiene $\overline{y} = 130.26$ digit e $\sigma_y = 0.53$ digit. Dato che la d.d.p. corrispondente era $\Delta V = (139.1 \pm 0.7)$ mV, si può dedurre, supponendo linearità, un fattore di calibrazione $\xi^* = (1.067 \pm 0.005)$ mV/digit, che è in ragionevole accordo con $\xi^*_{nom} = V_{ref}/1023 = 1.06$ mV/digit dedotto dal valore nominale $V_{ref} = 1.1$ V. Per questa impostazione della scheda Arduino non è stata eseguita la calibrazione tramite best-fit descritta in Sez. VII A.

IX. COMPORTAMENTI STRAVAGANTI

Il carattere stocastico delle singole misure rende naturalmente impossibile ottenere una completa ripetibilità dei campioni acquisiti. Qualche volta, e in maniera in genere non ripetibile, l'esito delle misure può discostarsi significativamente dalle aspettative [14]. La Fig. 6 riporta altri due esempi di campioni acquisiti per diversi valori di ΔV (l'impostazione di Arduino e la realizzazione dei grafici segue quanto riportato in Fig. 3). Si vede come talvolta la distribuzione occupi più di due bin (unitari) dell'istogramma, probabilmente a causa di una variazione di ΔV in ingresso verificatasi *durante* l'acquisizione del campione, e come altre volte il campione contenga misure *evidentemente* diverse da tutte le altre.

Una possibile motivazione per la registrazione di simili risultati è il cattivo funzionamento *temporaneo* del sistema sotto analisi (per esempio, le connessioni dei tanti spinotti e cavetti, oppure una piccola variazione della posizione dell'alberino del potenziometro). Quando si eseguono misure in maniera non automatica, per esempio guardando il display del tester, o lo schermo dell'oscilloscopio, ci si rende conto immediatamente di questi malfunzionamenti, e le misure corrispondenti vengono quindi immediatamente scartate. Questo non succede quando l'acquisizione è automatica (e la visualizzazione non è in tempo reale), e l'analisi anche grossolana dei risultati, compresa l'eventuale eliminazione di misure selezionate dal campione, può essere fatta solo a posteriori.

C'è poi un altro aspetto molto generale e altrettanto importante. Possiamo facilmente affermare che le fluttuazioni stocastiche della misura dipendono non soltanto dal comportamento dello strumento (qui, per noi, Arduino) ma anche dal comportamento del sistema, o circuito, sotto esame. Tuttavia la misura di grandezze elettriche è sempre, anche se in forma più o meno rilevante a se-

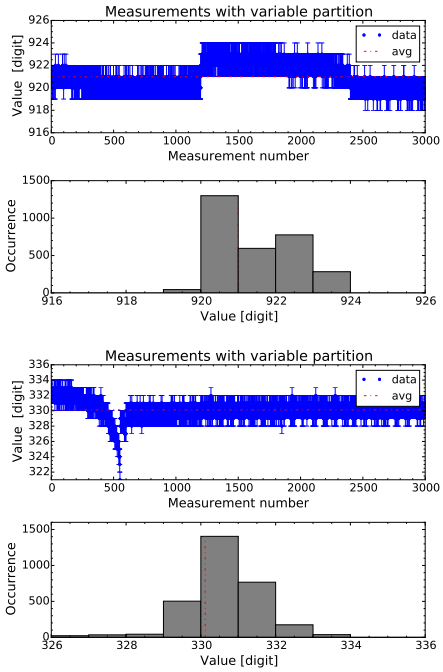

Figura 6. Altri esempi di campioni di misure ottenuti per diversi valori di ΔV. La rappresentazione segue quanto indicato in didascalia di Fig. 3.

conda dell'impiego di opportune tecniche, affetta da fluttuazioni che dipendono da cause *esterne* al sistema, o, più correttamente, dall'interazione di cause esterne con il sistema. Molto spesso, queste fluttuazioni entrano a far parte del cosiddetto *rumore* (anzi, molto spesso in una misura tutto quello che non è "segnale" si chiama "rumore").

Andando avanti con gli studi vedrete alcune classificazioni generali del rumore. Per il momento, e in modo molto grossolano, possiamo individuare delle ovvie cause di rumore i cui effetti possono facilmente sovrapporsi a quelli della misura progettata. Sicuramente esistono dei rumori *a bassa frequenza*, dovuti all'accoppiamento spurio del campo elettrico e, soprattutto, magnetico prodotto dalla corrente alternata (a 50 Hz, da cui la definizione "a bassa frequenza") che alimenta le varie apparecchiature. Altrettanto sicuramente esistono dei rumori *ad alta frequenza* legati al campo elettromagnetico presente negli ambienti in cui operiamo (stazioni radio, ma anche, e soprattutto, telefonini, e tutti gli oscillatori che stanno dentro ai dispositivi di uso comune, come computer, tablet, ancora telefonini, etc.). Anche questi campi elettromagnetici possono facilmente accoppiarsi ai cavi, spinotti e boccole che usiamo in laboratorio.

Della presenza di questi rumori vi renderete facilmente conto usando l'oscilloscopio. Probabilmente vedrete anche, in questo corso e nel futuro, che esistono opportune tecniche tese a limitarne gli effetti, per esempio grazie all'uso di cavi e connettori "schermati". Per il momento facciamo una facile osservazione: dei rumori normalmente ci disinteressiamo quando usiamo il tester digitale, almeno nella maggior parte delle situazioni pratiche. Questo è dovuto al fatto che il tester digitale mostra il risultato della misura avendo eseguito una sorta di integrazione temporale (la prontezza dello strumento, che è data dal tempo di refresh del display, è molto bassa). Molti rumori hanno un andamento temporale a media nulla, per cui i loro effetti sono cancellati se si aspetta (ovvero si integra su) un tempo sufficientemente lungo. Invece Arduino campiona in intervalli di tempo brevi (decine di microsecondi), dove l'effetto di alcuni tipi di rumore è esaltato.

Per illustrare questi aspetti può essere istruttivo costruire dei campioni di misure realizzati senza impiegare il partitore. La Fig. 7 mostra degli esempi relativi rispettivamente al campione costruito *senza collegare nulla* alle boccole di ingresso di Arduino, oppure collegando una resistenza di alto valore (6.8 Mohm nominali) tra pin `A0` e `GND`. Questi esempi mostrano che, in queste condizioni, si può ottenere di tutto. In particolare, a circuito disconnesso si osserva una sorta di oscillazione, o meglio a dei battimenti fra oscillazioni [15], forse legati al rumore a bassa frequenza, sovrapposta a una sorta di discesa esponenziale del valore misurato. Quest'ultima può essere interpretata come dovuta alla scarica del condensatore incluso nel circuito del digitalizzatore (sample and hold) attraverso la resistenza, estremamente alta, di ingresso del circuito. Collegando una resistenza da 6.8 Mohm, l'effetto di scarica non si osserva più, ma restano fluttuazioni molto rilevanti sempre probabilmente legate al rumore prodotto da cause esterne.

Della sensibilità delle misure nei confronti del rumore dovrete sempre, d'ora in avanti, essere ben consapevoli.

ACKNOWLEDGMENTS

La realizzazione delle esperienze con Arduino su tutti (o quasi) i banchi di laboratorio richiede una notevole mole di lavoro per l'installazione e il mantenimento dei computer e del software necessario. Queste esperienze non avrebbero potuto essere programmate se un'enorme mole di lavoro tecnico non fosse stata eseguita dal personale del Dipartimento di Fisica. In particolare si ringrazia qui tutto il personale tecnico dei Laboratori Didattici di Fisica I e II anno, Virginio Merlin e Carmelo Sgrò che, in assenza di Luca Baldini, ha brillantemente contribuito a risolvere alcuni fastidiosi problemi.

[1] In seguito a recenti sviluppi della situazione societaria di Arduino, esiste anche un'altra denominazione per la scheda, che è "Genuino". Per tradizione, continueremo

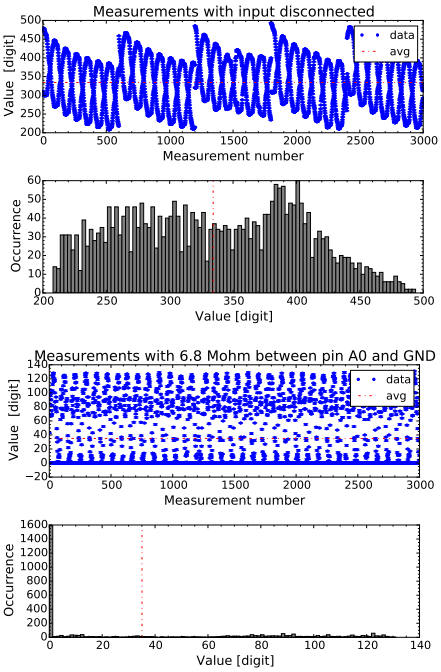

Figura 7. Altri esempi di campioni di misure ottenuti tenendo scollegati ("flottanti") gli ingressi di Arduino oppure collegando una resistenza di alto valore, 6.8 Mohm nominali, tra pin `A0` e `GND`. La rappresentazione segue quanto indicato in didascalia di Fig. 3.

comunque a usare il nome Arduino.

[2] Nell'elettronica digitale, gli stati, o livelli, alti o bassi corrispondono rispettivamente agli 0 e 1 della logica binaria. I valori di d.d.p. corrispondenti possono essere definiti secondo numerosi standard, il più comune dei quali è lo standard TTL (Transistor-Transistor Logic), che è anche impiegato per le porte digitali di Arduino. Esso fa corrispondere il livello basso a una tensione compresa tra 0 e 0.8 V, e il livello alto a una tensione superiore a 2 V. Nella pratica, l'implementazione di molti dispositivi, Arduino compreso, prevedono che il livello basso corrisponda a una d.d.p. circa nulla, e quello alto a una d.d.p. di circa 5 V.

[3] La situazione sarà diversa in esperienze future, nelle quali sarà indispensabile determinare l'istante di acquisizione delle singole misure.

[4] Il motivo è banale. Il microcontroller di Arduino, come qualsiasi CPU, ha ritardi, o tempi di latenza, non determinati durante l'esecuzione di un ciclo di programma. Infatti, oltre ad eseguire il programma, deve preoccuparsi di gestire il suo stesso funzionamento, cioè di controllare, per esempio, lo stato della memoria, la sua alimentazione, la presenza di segnali sulle porte, l'accensione o lo spegnimento di segnali interni ("interrupts"), etc.. Di conseguenza il controllo sul timing delle varie operazioni, inclusi i cicli di ritardo inseriti nel programma, è affetto da incertezza.

[5] Il tempo effettivo di campionamento non è un parametro noto dai datasheet del microcontroller. Secondo diverse fonti, esso è dell'ordine delle (poche) decine di microsecondi.

[6] Per favore, tenete conto che il potenziometro è, per sua natura, un dispositivo delicato e poco affidabile. Infatti il contatto strisciante può facilmente funzionare in modo non corretto, dando luogo a una resistenza diversa da quella attesa per una data posizione dell'alberino. Inoltre in certe condizioni è sufficiente sfiorare la manopola fissata sull'alberino per ottenere variazioni poco controllate della resistenza.

[7] A parte gli ovvi motivi di disponibilità di tempo e di capacità di elaborazione dati, ci sono buone ragioni per evitare acquisizioni troppo lunghe, cioè ripetute su più di *qualche* ciclo. Esse risiedono principalmente nel fatto che il sistema qui impiegato, come la maggior parte dei sistemi fisici, soffre di variazioni delle condizioni di funzionamento (*drifts*) a medio termine, per esempio sulla scala dei minuti. Queste variazioni possono essere legate a diverse cause fisiche: di norma, per i nostri esperimenti la principale è la variazione di temperatura, che si sviluppa tipicamente su queste scale temporali. Di conseguenza è ovvio che le dimensioni del campione possono influenzare la "bontà" di ricostruzione della distribuzione. Normalmente la presenza di fluttuazioni è più evidente se si usa un campione piccolo e d'altra parte la deviazione standard sperimentale generalmente cala se il campione è grande.

[8] Come sarà accennato nel seguito, la digitalizzazione è esposta a numerose cause di errore legate alla presenza di segnali spurii sul pin di lettura. Molti segnali spurii hanno un'origine "impulsiva", cioè sono legati a degli impulsi. All'inizio dell'acquisizione ci sono numerosi impulsi in giro per la scheda di Arduino (per esempio, tutti i comandi che vengono forniti al microcontroller), che possono sicuramente accoppiarsi con l'ingresso analogico. Scartare la prima misura è una buona prescrizione di sicurezza, da applicare quando possibile.

[9] Visto come è realizzato lo sketch, è evidente il vantaggio di avere campionamenti nominalmente equispaziati nel tempo.

[10] Ricordiamo anche che l'eventuale mancanza di precisione nella temporizzazione delle misure non costituisce alcun problema per la presente esperienza.

[11] È possibile alimentare la scheda Arduino indipendentemente dalla porta USB usando un alimentatore.

[12] L'andamento lineare con un offset è in effetti quanto riportato nella documentazione del microcontroller ATMega 328 P installato in Arduino Uno.

[13] Impostando la scheda Arduino per usare il riferimento interno $V_{ref} = 1.1$ V nominali non si modifica la massima d.d.p. erogabile dalle porte digitali, per cui il metodo alternativo di calibrazione mostrato in Sez. VII non è più applicabile. Può essere interessante, in prospettiva futura, sapere che questa referenza interna è detta tensione di *energy gap*. Infatti essa, che è creata attraverso un ingegnoso circuito, è pari all'energy gap del Silicio, una grandezza che conoscerete studiando il comportamento dei materiali semiconduttori (grazie a Diego per l'illuminazione).

[14] Per molte schede Arduino, comportamenti particolarmente "critici" si osservano spesso in corrispondenza di letture attorno ai valori 128, 256, 512, 768 (tutti esprimibili con potenze di 2). In certe condizioni si verifica infatti che Arduino mostra una specie di inerzia nel seguire andamenti transienti, cioè nel digitalizzare valori di d.d.p. che cambiano nel tempo. In questi casi l'incertezza effet-

tiva può diventare sensibilmente maggiore di ±1 digit. I motivi sono probabilmente legati alla specifica architettura interna del digitalizzatore integrato nel microcontroller. Fortunatamente questi problemi non compaiono, normalmente, quando si opera in continua, cioè quando il segnale analizzato non ha carattere transiente.

[15] I battimenti sono fenomeni di sovrapposizione tra due o più perturbazioni periodiche, o quasi-periodiche, che danno luogo a strutture tipiche composte da oscillazioni ad "alta frequenza" (proporzionale alla somma delle frequenze delle due o più perturbazioni) convolute con oscillazioni a "bassa frequenza" (proporzionale alla differenza delle frequenze delle due o più perturbazioni). La rappresentazione usata in Fig. 7 non permette di visualizzare le oscillazioni ad alta frequenza a causa della risoluzione temporale impiegata. Infatti in essa è possibile scorgere solo l'"'inviluppo"dei battimenti, che sembra dare luogo a una doppia oscillazione (la figura si riferisce all'acquisizione di una sola traccia, due distinte oscillazioni non possono essere acquisite). Espandendo la scala, e usando una rappresentazione per linee invece che per punti, si possono osservare oscillazioni rapide, con un periodo apparente paragonabile al rate di campionamento (nell'esempio considerato, 10 ms nominali). È possibile che, in corrispondenza della digitalizzazione, all'interno di Arduino si formino degli impulsi elettrici, che di fatto costituiscono un rumore quasi-periodico, a frequenza simile al rate di campionamento. Il battimento potrebbe allora essere dovuto all'interferenza fra questo rumore e il rumore dovuto alla presenza di campi magnetici alternati alla frequenza di rete (50 Hz), accoppiati per *pick-up* all'ingresso analogico di Arduino.