Arduino e campionamento/digitalizzazione di segnali continui

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

(Dated: version 5 - FF, 25 ottobre 2016)

Queste nota ha un duplice scopo: in primo luogo essa presenta alcune generalità sulle modalità di impiego di Arduino, discusse 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 [1], realizzata in laboratorio allo scopo di calibrare il digitalizzatore e di stimare le incertezze di misura.

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 *campione* disponibile per analisi statistiche (calcolo della media, della deviazione standard, istogrammi delle occorrenze, etc.), oltre a permettere di eseguire una calibrazione del digitalizzatore.

Calibrazione a parte, dal punto di vista pratico l'operazione descritta ha poco senso: infatti in genere è utile "acquisire" digitalmente d.d.p. che variano nel tempo, dato che dall'analisi dell'andamento temporale possono essere dedotte importanti informazioni sui sistemi, o i circuiti, sotto analisi. Proprio in previsione dell'impiego con segnali transienti e periodici, oltre al campione di d.d.p. digitalizzate si costruisce un campione rappresentativo degli istanti di digitalizzazione. Gli aspetti che riguardano più strettamente la tempistica di acquisizione, in particolare quelli connessi al cosiddetto *campionamento* dei dati, saranno trattati più estesamente in altre esperienze e altre note. Qui la conoscenza del tempo di digitalizzazione ci servirà solo per stimare l'incertezza associata.

Per ottenere gli scopi della presente esperienza 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 produrranno in modo automatico i campioni di nostro interesse, registrati in un file di due colonne (tempo e valore digitalizzato) disponibile per le ulteriori analisi.

Poiché, come chiariremo nel seguito, il trasferimento dei dati da Arduino al computer è generalmente lento (richiede secondi), e visto 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 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 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, in genere, essendo identificato dall'icona di Arduino (un infinito su sfondo verde acqua). 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 disponibili devono essere considerate come ingressi e uscite, quelle che eseguono la lettura di una porta analogica o

la "scrittura" (determinazione di livello "alto" o "basso" [2]) 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, e le variabili previste (tipicamente array) devono avere piccole dimensioni. Una volta che lo sketch compilato è stato trasferito, 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.

Le misure della nostra esperienza, non essendo legate a variazioni temporali, e quindi ad eventi che si verificano in determinati istanti, possono essere considerate *asincrone*. Infatti la grandezza da misurare è ritenuta continua, per cui non c'è alcun motivo che spinga ad acquisire i dati in istanti specifici [3]. È tuttavia sempre consigliabile avere una qualche forma di controllo sulla partenza delle operazioni compiute da Arduino. 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 USB. 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. 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 confusione);

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 anche in modo automatico, per esempio per verificare la presenza di fluttuazioni o per analizzare il comportamento di diverse configurazioni sperimentali, come discusso nel seguito.

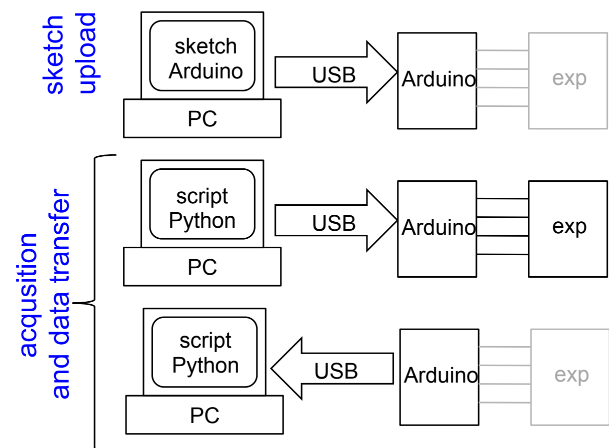

Figura 1. Diagramma logico di massima di una tipica esperienza con uso di Arduino.

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 bit: 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 bit, e che ora è estesa, potenzialmente, su ben 21 bit. 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 nativamente nei computer di laboratorio, cioè quella che si richiama da terminale), 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 da 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. Infatti la dinamica, o profondità di digitalizzazione, di Arduino è 10 bit, ovvero i livelli di digitalizzazione possibili sono $2^{10} = 1024$.

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. Tuttavia, come preparazione a ulteriori esperienze con Arduino, e anche allo scopo di studiare l'incertezza della misura dei tempi, anche nella presente esperienza Arduino è predisposto per acquisire il *time stamp*, cioè l'indicazione dell'"istante" di digitalizzazione (riferito naturalmente a un tempo zero, opportunamente definito). 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: l'analisi dei dati corrispondenti permetterà di verificare entro quale accuratezza l'impostazione nominale (equispaziatura) è effettivamente realizzata.

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. la digitalizzazione non può essere istantanea e il campionamento avviene in un breve intervallo di tempo che è 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 successive [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* collegato al solito alimentatore che siete ormai abituati ad usare. Per evitare di essere

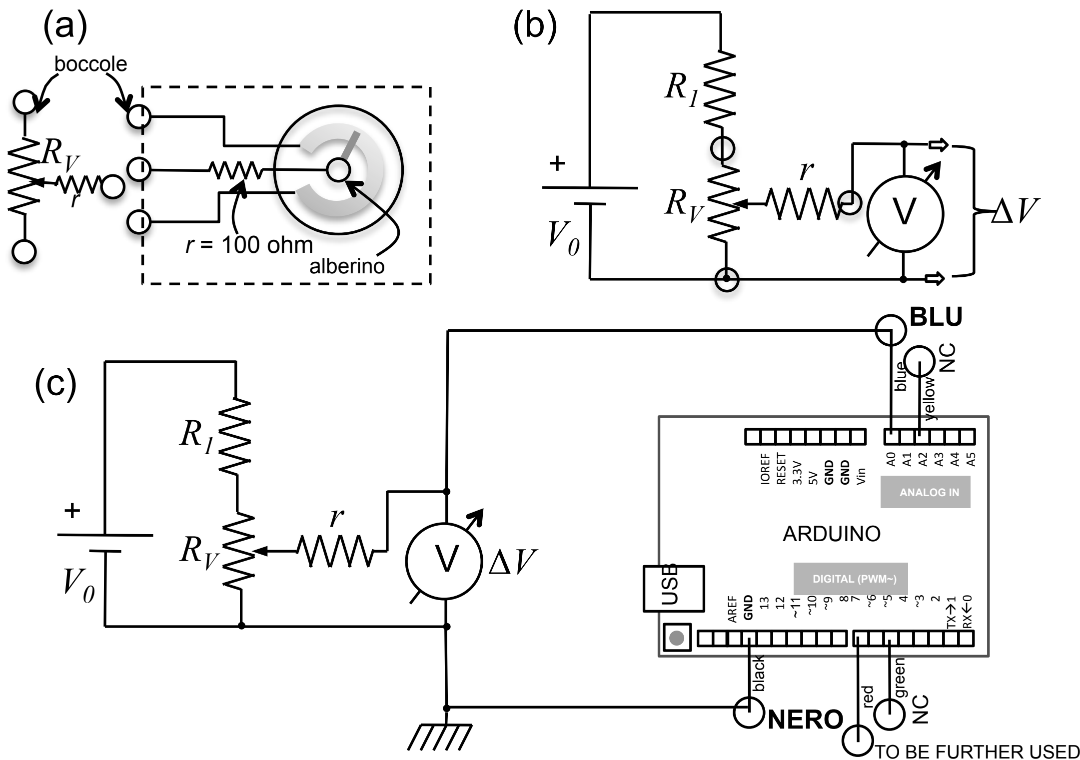

Figura 2. Rappresentazione schematica e costruttiva del potenziometro "visto da sotto" (a), schema del partitore di tensione (b), configurazione del circuito di misura comprendente Arduino (c). Nel pannello (a) è riportato uno dei simboli con cui si indica un potenziometro negli schemi elettronici. Nel 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 (il collegamento a terra è realizzato attraverso l'alimentatore del PC di laboratorio).

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 il contatto strisciante stesso e le due 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 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 [6]) 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_V (potete facilmente dimostrarlo con le regoline dei partitori di tensione) [7]. 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 e dunque "perturba" in modo trascurabile il circuito. 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). 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 e passare nello stato di big sleep.

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 quella rossa, collegata alla porta digitale corrispondente al pin `7`, da usare per gli scopi di calibrazione (alternativa) che descriveremo in seguito. Le connessioni da effettuare, esclusa quella eventuale al pin `7`, sono schematizzate in Fig. 2(c).

Come informazione rilevante dal punto di vista pratico, ricordate che Arduino mantiene i programmi nella sua memoria flash finché questi non vengono riscritti. Di conseguenza è possibile che, collegando Arduino al circuito come in Fig. 2 senza aver preventivamente fatto l'upload dello sketch di interesse, il comportamento del circuito sia erroneo. Dunque come norma collegate Arduino *solo dopo aver effettuato l'upload dello sketch.*

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 256 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 [8]. Questo loop è avviato dall'istruzione `for j in range (1,nacqs+1):`, con `nacqs` da definire nello script (di default pari a 1). 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, esattamente come nella definizione di funzioni. Inoltre nella parte iniziale dello script si stabilisce la directory che conterrà i dati. Di default, alla directory dove sono raccolti i dati nei computer di laboratorio si accede con `../dati_arduino/` (si intende che Python sia lanciato avendo `Home` come directory presente). È anche necessario fornire il nome del file, che dovrete stabilire secondo i vostri gusti, compresa l'estensione (consigliata) `.txt`.

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 fortemente dipendente dal sistema operativo) e specificato che la comunicazione avverrà alla velocità di 19200 Baud (nella comunicazione usata, un Baud dovrebbe corrispondere a 1 bit/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'5')` 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 '5'.

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_{nom} tra una digitalizzazione e la successiva. L'unità di misura è, in questa esperienza, 100 μs, per cui il "5" significa che i dati saranno campionati con intervalli nominali di $5 \times 100\ \mu\text{s} = 500\ \mu\text{s}$. La scelta di questo parametro non influenza i risultati della presente esperienza, mentre invece sarà critica per altre esperienze da svolgere in futuro. Essa sarà inoltre considerata in seguito, quando tratteremo dell'analisi dell'incertezza del tempo di digitalizzazione.

Quindi lo script attende finché sulla porta seriale, continuamente monitorata, non compaiono dei dati. Ardui-

no 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 (256, nell'esempio considerato). Notate la sintassi abbastanza specifica per l'operazione di lettura di Arduino: essa prevede di leggere una riga (coppia di dati) 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. Per agevolare alcune delle operazioni previste nell'esperienza, lo script si occupa anche di calcolare valore medio e deviazione standard *sperimentale* del campione di dati digitalizzati (si intende campione di 256 punti, in questo esempio). Visto che, come sarà chiarito in seguito, i dati vengono codificati da Arduino nella forma di righe (per un totale di 256, nell'esempio qui considerato) contenenti il time stamp in unità di μs, uno spazio, il valore digitalizzato (in digit), occorre un'istruzione dalla sintassi apparentemente misteriosa, `runningddp[i]=data[data.find(' '):len(data)]`, per estrarre il dato di interesse e metterlo in un array di supporto. Quindi media e deviazione standard sono calcolate su questo array usando istruzioni standard di Python e il risultato viene scritto sulla console. Alla fine di tutto compare un bell'`end`.

Lo script, debitamente commentato, è riportato qui di seguito; esso si trova nei computer di laboratorio (nella directory `/Arduini/`) e in rete sotto il nome di `ardu2016.py`.

```
import serial # libreria per gestione porta seriale (USB)
import time # libreria per temporizzazione
import numpy

nacqs = 1 # numero di acquisizioni da registrare (ognuna da 256 coppie di 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 predisposto per scrittura

for j in range (1,nacqs+1):
    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'5') # scrive il carattere per l'intervallo di campionamento
                    # in unita' di 100 us << DA CAMBIARE A SECONDA DEI GUSTI
                    # l'istruzione b indica che è un byte (carattere ASCII)
    time.sleep(2) # aspetta due secondi per evitare casini
    print('Start Acquisition ',j, ' of ',nacqs) # scrive sulla console (terminale)
    # loop lettura dati da seriale (256 coppie di dati: tempo in us, valore digitalizzato di d.d.p.)
    runningddp=numpy.zeros(256) # prepara il vettore per la determinazione della ddp media e std

    for i in range (0,256):
        data = ard.readline().decode() # legge il dato e lo decodifica
        if data:
            outputFile.write(data) # scrive i dati sul file
            runningddp[i]=data[data.find(' '):len(data)] # estrae le ddp e le mette nel vettore
    ard.close() # chiude la comunicazione seriale con Arduino

    avgddp=numpy.average(runningddp) # analizza il vettore per trovare la media
    stdddp=numpy.std(runningddp) # e la deviazione standard
    print('Average and exp std:', avgddp, '+/-',stdddp) # le scrive sulla console

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 al C. In questo linguaggio si fa uso molto spesso di pseudo-funzioni, cioè gruppi di istruzioni che non ri-

tornano 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. Esso è di fatto suddiviso in tre parti poste consecutivamente una dietro l'altra: (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 e l'allocazione dello spazio di memoria relativo 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.

Nel nostro caso abbiamo sicuramente a che fare con due distinte variabili di tipo array, quelle che vanno acquisite e registrate. Esse sono l'array denominato `V`, che contiene il valore digitalizzato della d.d.p., e quello denominato `t`, che contiene il time stamp. Nel primo caso è sufficiente definire l'array come intero a singola precisione, `int`, visto che il valore digitalizzato è necessariamente compreso tra 0 e 1023. Nel secondo caso, invece, visto che il time stamp è in unità di microsecondi e che la durata complessiva dell'acquisizione può essere "lunga" su questa scala, occorre definire la variabile come `long`: infatti, per l'intero campione di 256 misure, se per esempio l'intervallo di campionamento nominale è $\Delta t_{nom} = 500$ μs, l'ultima misura avviene (almeno) dopo 1.28×10^3 μs, numero per la cui registrazione non basta un intero a singola precisione. Le due istruzioni di definizione sono rispettivamente `int V[256];` e `long t[256];`, le quali mostrano pure che gli array sono entrambi costituiti da 256 punti.

Il resto di questa sezione dello sketch, che è riportata qui nel seguito, è piuttosto auto-esplicativa: notate che `analogPin` e `digitalPin` 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 alternativa", cioè le porte corrispondenti ai pin `A0` e `7` della scheda. 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 `delays` contiene il ritardo nominale Δt_{nom} (in μs) tra una digitalizzazione e la successiva. Infine, la variabile `StartTime`, definita `long`, serve per indicare lo zero dei tempi, secondo quanto sarà chiarito in seguito.

```
// Blocco definizioni
const unsigned int analogPin=0; // Definisice la porta A0 per la lettura
const int digitalPin=7; // Definisce la porta 7 usata come output ref
int i; // Definisice la variabile intera i (contatore)
int delays; // Definisce la variabile intera delays
int V[256]; // Definisce l'array intero V
long t[256]; // Definisice l'array t
unsigned long StartTime; // Definisce la variabile StartTime
int start=0; // Definisce la variabile start (usato come flag)
```

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 19200 Baud (`Serial.begin(19200);`), di pulirne per sicurezza il buffer (`Serial.flush();`), di definire di uscita la porta indicata dalla variabile `digitalPin` e di porla a livello "alto", per gli scopi di cui tratteremo in seguito (l'istruzione è auto-esplicativa). Notate che non è necessario definire la porta analogica come input, essendo

questa la configurazione di default.

Inoltre il blocco di inizializzazione contiene due linee di istruzione tanto misteriose quanto utili (le spiegazioni relative, non date qui, si trovano facilmente in rete): esse consentono ad Arduino di operare la digitalizzazione (quasi) al massimo del rete di campionamento possibile, che è dell'ordine di alcune (poche) decine di μs. Senza entrare troppo nei dettagli, il loro scopo è di istruire i registri interni al microcontroller in modo che esso sia in un certo senso "over-clocked" rispetto alle condizioni ordinarie. Naturalmente per l'esperienza presente questa specifica è del tutto inutile, visto che non serve a niente campionare "ad alta velocità" un segnale costante, però essa è mantenuta per analogia con quanto faremo nella maggior parte dei nostri impieghi sperimentali di Arduino.

La parte di sketch che riguarda l'inizializzazione è la seguente:

```
// Istruzioni di inizializzazione
void setup()
  {
   Serial.begin(19200); // Inizializza la porta seriale a 19200 baud
   Serial.flush(); // Pulisce il buffer della porta seriale
   digitalWrite(digitalPin,HIGH); // Pone digitalPin a livello alto
   bitClear(ADCSRA,ADPS0); // Istruzioni necessarie per velocizzare
   bitClear(ADCSRA,ADPS2); // il rate di digitalizzazione
  }
```

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 ritorna 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 100, deve dare l'intervallo nominale in μs tra due istanti successivi di campionamento. Il *numero* di μs 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 ($53 - 48 = 5$). Esso poi va, come detto, moltiplicato per 100 allo scopo di definire la variabile `delays`, che contiene il numero di μs che deve nominalmente intercorrere tra una digitalizzazione e la successiva.

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 2000 ms attraverso l'istruzione `delay(2000)` (l'unità di misura è qui ms). Questa pausa tranquillizza nei confronti di possibili intasamenti nel funzionamento del microcontroller (credo sia possibile ridurne la durata in caso di necessità).

A questo punto inizia il ciclo di misure. La digitalizzazione del segnale sulla porta di ingresso indicata dalla variabile `analogPin` avviene attraverso l'istruzione `analogRead(analogPin);`, che ritorna un intero compreso tra 0 e 1023. Notate che, prima di iniziare le misure "vere e proprie" (quelle che vengono effettivamente registrate), lo sketch prevede un ciclo di due misure a vuoto. Lo scopo è di minimizzare l'acquisizione di *artefatti*, cioè misure falsate, dovuti alla presenza segnali spuri generati all'interno di Arduino all'inizio delle misure. Il problema non è atteso avere effetti rilevanti per la presente esperienza, ma conviene prevedere le due misure "a vuoto" per uniformità con quanto faremo in futuro.

Quindi, subito prima di avviare il ciclo di misure vere e proprie (da registrare), Arduino misura il suo tempo interno (in unità di μs) e lo mette nella variabile `StartTime`, usando l'istruzione `StartTime=micros();`; il tempo interno scorre ciclicamente a partire dall'istante in cui il programma è stato lanciato e il valore che viene qui registrato verrà poi sottratto alle misure di tempo eseguite in corrispondenza delle digitalizzazioni, in modo da

costruire un time stamp riferito sempre (nominalmente) all'inizio delle acquisizioni.

Finalmente ha inizio il ciclo di misure, che, in questo esempio, si svolge su 256 digitalizzazioni. L'istruzione che avvia il ciclo è `for(i=0;i<256;i++)`, con ovvia sintassi; le istruzioni del ciclo sono comprese tra parentesi graffe. Il risultato delle misure va negli elementi i-esimi degli array `V[i]`, grazie all'istruzione `V[i] = analogRead(analogPin);`, e `t[i]`, grazie all'istruzione `t[i]=micros()-StartTime;`, che, come detto prima, determina il tempo a partire dall'istante iniziale immagazzinato in precedenza nella variabile `StartTime`. Queste istruzioni sono seguite dalla `delayMicroseconds(delays);`, che temporizza il campionamento su intervalli distanti temporalmente per il valore impostato [9]. Attraverso quanto descritto in seguito verificheremo la corrispondenza tra valore effettivo e nominale di tale intervallo.

Terminata l'acquisizione dei 256 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 files in formato testo che possono facilmente essere letti da Python. I dati sono organizzati in righe contenenti, nell'ordine, il valore i-esimo dell'array `t[i]`, uno spazio, il valore i-esimo dell'array `V[i]`. Al termine di ciascuna riga si "va a capo" per iniziarne una nuova; questo è realizzato dall'istruzione `Serial.println(V[i]);` (l'istruzione che serve per scrivere i dati senza andare a capo è `Serial.print`).

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 e nei computer di laboratorio sotto il nome `ardu2016.ino`):

```
// Istruzioni del programma
void loop()
  {
    if (Serial.available() >0) // Controlla se il buffer seriale ha qualcosa
      {
        delays = (Serial.read()-'0')*100; // Legge il byte e lo interpreta come ritardo
        Serial.flush(); // Svuota la seriale
start=1; // Pone il flag start a uno
      }
  if(!start) return // Se il flag e' start=0 non esegue le operazioni qui di seguito
                    // altrimenti le fa partire (quindi aspetta di ricevere l'istruzione
                    // di partenza
    delay(2000); // Aspetta 2000 ms per evitare casini
     for(i=0;i<2;i++) // Fa un ciclo di due letture a vuoto per "scaricare" l'analogPin
       {
        V[i]=analogRead(analogPin);
       }
    StartTime=micros(); // Misura il tempo iniziale con l'orologio interno
    for(i=0;i<256;i++) // Loop di misura
      {
          t[i]=micros()-StartTime; // Legge il timestamp e lo mette in array t
          V[i]=analogRead(analogPin); // Legge analogPin e lo mette in array V
          delayMicroseconds(delays); // Aspetta tot us
      }
    for(i=0;i<256;i++) // Loop per la scrittura su porta seriale
      {
        Serial.print(t[i]); // Scrive t[i]
        Serial.print(" "); // Mette uno spazio
        Serial.println(V[i]); // Scrive V[i] e va a capo
      }
    start=0; // Annulla il flag
    Serial.flush(); // Pulisce il buffer della porta seriale (si sa mai)
}
```

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, e un campione di misure degli istanti a cui la grandezza è sta-

ta digitalizzata. Anche se l'analisi non è particolarmente significativa dal punto di vista della fisica coinvolta, questi campioni possono essere trattati con metodi statistici (costruzione dell'istogramma delle occorrenze, calcolo di media e deviazione standard sperimentale). Lo scopo principale è quello di stimare l'incertezza da associare alle misure di d.d.p. digitalizzata e di tempo condotte da Arduino nelle condizioni tipiche dei nostri esperimenti.

Dunque qui di seguito vengono riportati esempi e commenti, rimandando alle sezioni successive per la discussione delle operazioni di calibrazione del digitalizzatore (che, invece, sono argomento principale dell'esperienza pratica di laboratorio).

A. Campione di misure digitalizzate

La Fig. 3 mostra un esempio di campione ottenuto registrando 8 blocchi consecutivi di 256 misure (usando `nacqs=8` nello script di Python), per un totale di 2048 dati, in presenza di una d.d.p. $\Delta V = (2.65 \pm 0.02)$ V, misurata con il multimetro digitale. La lettura del multimetro 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 del tutto trascurabili.

Il pannello superiore riporta il valore della d.d.p. digitalizzata (dunque l'unità di misura è digit, secondo la nostra convenzione) in funzione del numero progressivo della misura. Ai dati sperimentali è associata un'*incertezza convenzionale* pari a ± 1 digit. Come si vede, la misura è stabile: l'intero campione è infatti costituito dal valore 540 digit, con deviazione standard sperimentale nulla.

Per completezza, il pannello inferiore mostra l'*istogramma delle occorrenze* dello stesso campione: si vede che un solo bin, quello corrispondente alla lettura 540 digit, è popolato. 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 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: infatti è sicuro che bin "frazionari" hanno popolazione nulla. Inoltre, essendo interessati all'istogramma delle occorrenze, *non* è necessario, e anzi è sconsigliato, arrangiarsi per ottenere l'istogramma delle frequenze (normalizzate).

La situazione considerata, in cui si registra una deviazione standard sperimentale nulla, è piuttosto comune. In linea di principio, in un esperimento come quello qui condotto ci si aspetta che le misure abbiano una qualche

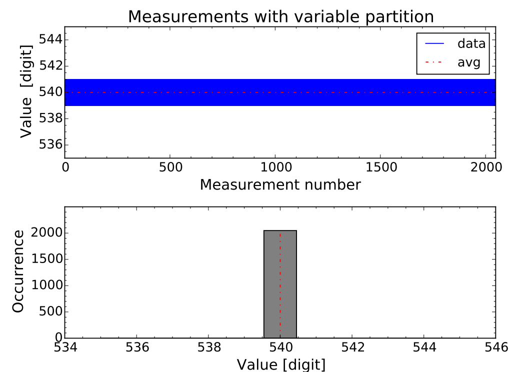

Figura 3. Esempio di analisi del campione di dati acquisito per $\Delta V = (2.65 \pm 0.02)$ V, misurata con il multimetro digitale. Il pannello superiore riporta il grafico delle misure, a cui è stata attribuita un'incertezza convenzionale ± 1 digit, il pannello inferiore riporta l'istogramma delle occorrenze. Il valore medio digitalizzato è 540 digit e la deviazione standard sperimentale è nulla, secondo quanto descritto nel testo.

distribuzione, che presumibilmente approssima la distribuzione normale (Gaussiana) all'aumentare delle dimensioni del campione. Qui la distribuzione non può essere ricostruita e questo si verifica semplicemente perché la sensibilità della misura (il valore fisico che corrisponde al singolo digit) non è sufficiente. Evidentemente, non è possibile stimare l'incertezza dalla "misura" (della media) dalla deviazione standard sperimentale, che ha un valore inferiore alla massima sensibilità. Di conseguenza scegliamo di individuare un'incertezza arbitraria, che convenzionalmente poniamo pari a ± 1 digit. Notate che, così facendo, possiamo ancora sostenere come tale incertezza abbia un'origine prevalentemente statistica, ma sicuramente ne stiamo compiendo una sovrastima. D'altra parte, come avremo modo di verificare in ulteriori esperienze, il digitalizzatore di Arduino può anche avere incertezze di origine sistematica, specialmente quando impiegato per acquisire d.d.p. variabili nel tempo.

Ricordate poi che la lettura automatizzata di dati non esercita (non può esercitare, a meno di introdurre opportune strategie) alcuna forma di controllo sui valori effettivamente letti. È possibile che alcune delle letture eseguite da Arduino siano falsate da artefatti di varia natura, per esempio da "disturbi" sull'alimentazione o nello stadio di ingresso del digitalizzatore [10]. Queste misure falsate contribuiscono all'istogramma, pur essendo non necessariamente rappresentative della distribuzione Gaussiana che ci aspetteremmo (un esempio è mostrato nella sezione seguente). È interessante notare che la frequenza delle misure falsate può dipendere dalle condizioni di funzionamento di Arduino: negli esempi mostrati in questa nota, Arduino era collegato a un computer portatile alimentato a batteria. L'uso dei computer di laboratorio, alimentati dalla rete (e collegati alla linea di terra) può sicuramente aumentare la frequenza degli artefatti.

1. Operazione con riferimento interno a 1.1 V

Normalmente Arduino lavora usando una tensione di riferimento $V_{ref} \sim 5$ V generata a partire da quella di alimentazione (l'alimentazione nel nostro caso è fornita dalla presa USB del computer), servendosene per determinare la digitalizzazione. Su questo argomento torneremo in seguito, quando affronteremo la calibrazione del digitalizzatore. Tuttavia è ovvio che V_{ref} determina il valore massimo della d.d.p. che può essere digitalizzata, e di conseguenza anche la sensibilità dello strumento.

Esiste un'opzione, che si richiama mettendo nel blocco di inizializzazione dello sketch l'istruzione `analogReference(INTERNAL);`, che ordina ad Arduino di impiegare come riferimento una tensione generata internamente, di valore nominale $V_{ref} = 1.1$ V (l'incertezza non è nota). Evidentemente l'uso di questa istruzione modifica la sensibilità del digitalizzatore, che diventa di circa 1 mV, 5 volte minore di quella "ordinaria". Questa possibilità apre la strada per ottenere un campione con qualche distribuzione osservabile.

L'opzione è attivata nello sketch `ardu_1V1_2016.ino` disponibile in rete e nei computer di laboratorio. Ovviamente se si usa questo sketch occorre *tassativamente* che la d.d.p. in ingresso alla porta analogica `A0` sia $\Delta V \leq 1.1$ V: ciò si ottiene, per esempio, inserendo $R_1 = 6.8$ kohm (nominali) nel circuito di Fig. 2.

La Fig. 4 mostra un esempio di campione ottenuto con questo sketch, usando $\Delta V = (964 \pm 5)$ mV: si vede come questa volta la sensibilità della misura sia sufficiente per apprezzare delle variazioni, anche se la distribuzione suggerita dall'istogramma, fortemente asimmetrico, non assomiglia affatto a una Gaussiana. Per l'esempio considerato si ottiene un valore medio dei conteggi di 904.6 digit e una deviazione standard sperimentale di 7.6 digit. La "strana" distribuzione registrata potrebbe essere dovuta alla presenza di artefatti, secondo quanto accennato in precedenza.

Come ultima osservazione molto rilevante (e altrettanto ovvia), notiamo che in questi esperimenti la d.d.p. da misurare è ritenuta costante, ma non si ha nessuna garanzia che essa lo sia. In altre parole, le fluttuazioni registrate nel campione potrebbero avere luogo anche nel circuito (alimentatore e partitore) usato per generare la d.d.p. stessa, come è molto probabile che si verifichi per esempio a causa delle già accennate "limitazioni" del potenziometro.

B. Campione degli intervalli di campionamento

Le acquisizioni di questa esperienza si prestano anche a un altro tipo di analisi. Infatti possiamo esaminare la misura dei tempi come effettuata da Arduino (registrata in unità di μs nella prima colonna del file ottenuto) allo scopo di determinare l'intervallo effettivo di campionamento Δt, cioè l'intervallo temporale tra due misure successive. Lo scopo è quello di confrontare il risultato

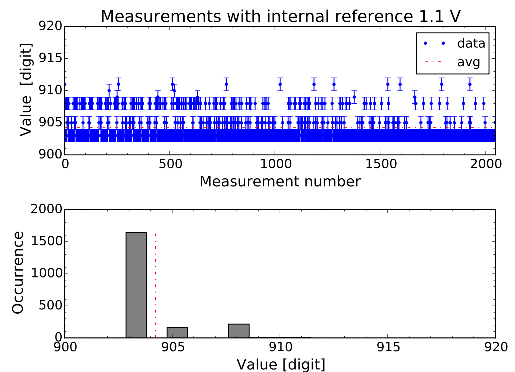

Figura 4. Analogo di Fig. 3 ottenuto utilizzando la tensione di riferimento interna $V_{ref} = 1.1$ V (nominali) e misurando con il multimetro digitale $\Delta V = (964 \pm 5)$ mV. Il valore medio digitalizzato è 904.6 digit e la deviazione standard sperimentale è 7.6 digit.

con il valore nominale Δt_{nom} impostato nello script di Python (in questo esempio, $\Delta t_{nom} = 500$ μs).

Dalle specifiche di Arduino, si sa che i tempi sono determinati, e quindi anche misurati, con una risoluzione di 4 μs, legata alla accuratezza del clock. Dunque appare ragionevole impiegare in prima battuta un'incertezza convenzionale $\delta t = 4$ μs per la misura dei tempi. L'analisi del campione ci consentirà di verificare la validità di questa scelta, secondo quanto discusso nel seguito.

La Fig. 5 mostra nel pannello superiore gli intervalli di tempo misurati Δt in funzione del numero progressivo di misura, corredati dell'incertezza appena specificata; il pannello inferiore rappresenta l'istogramma delle occorrenze per il campione considerato. Notate che produrre il campione dal file è un'operazione non del tutto banale: infatti, detti t_j i tempi misurati da Arduino, da ogni blocco di 256 misure possono essere estratti $(256 - 1)$ valori $\Delta t_j = t_{j+1} - t_j$, e la produzione di un unico array contenente i dati per l'intera acquisizione richiede attenzione nello scrivere un corretto algoritmo (questa parte può sicuramente essere considerata facoltativa nell'esperienza pratica).

Come si osserva in figura, l'intervallo Δt è diverso dall'impostazione nominale $\Delta t_{nom} = 500$ μs: il valore medio ottenuto è infatti 515.4 μs. La discrepanza può facilmente essere attribuita ai tempi di latenza del microcontroller e, soprattutto, al tempo minimo effettivo necessario affinché la digitalizzazione sia conclusa, che, dalle misure, risulta di circa 15 μs. È infatti ragionevole supporre che nel tempo necessario alla digitalizzazione Arduino sospenda le altre operazioni, in particolare quelle di temporizzazione. Inoltre la deviazione standard sperimentale ottenuta dal campione è 3.5 μs: questo valore può essere considerato come un'indicazione dell'incertezza associata alla misura del tempo da parte di Arduino nelle condizioni della nostra esperienza. Notate tuttavia che la distribuzione mostrata nell'istogramma è anche in questo caso tutt'altro che Gaussiana. Il risultato suggerisce comun-

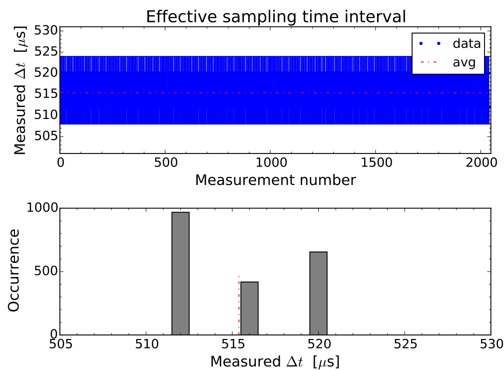

Figura 5. Esempio di analisi del campione di intervalli di campionamento Δt costruito come discusso nel testo. Il pannello superiore riporta il grafico delle misure, a cui è stata attribuita un'incertezza convenzionale ± 4 μs, il pannello inferiore riporta l'istogramma delle occorrenze. Il valore medio digitalizzato è 515.4 μs e la deviazione standard sperimentale è 3.5 μs.

que che l'incertezza dedotta dalle specifiche (4 μs) è simile (leggermente superiore, nel nostro esempio) a quella determinata esaminando il campione.

VII. CALIBRAZIONE DI ARDUINO

Nelle nostre esperienze Arduino viene impiegato principalmente come misuratore di d.d.p.: anche se in qualche occasione sarà sufficiente per i nostri scopi conoscere le tensioni misurate in unità arbitrarie di digitalizzazione (digit), spesso sarà necessario convertire tali unità in unità fisiche (V). Per questo è necessario eseguire una *calibrazione*.

In termini generali, è evidente che la calibrazione dipende dalle caratteristiche della specifica scheda Arduino impiegata e dal valore di V_{ref} che, in condizioni di operazione ordinarie (senza cioè scegliere il riferimento interno da 1.1 V nominali), dipende dall'alimentazione che Arduino riceve tramite USB. Dunque in linea di principio la calibrazione andrebbe ripetuta a ogni impiego di Arduino. Fortunatamente, come specificheremo nella prossima sezione, esiste un modo approssimato e molto rapido (*calibrazione alternativa*) per determinare un fattore di calibrazione, cioè di conversione tra digit e V. Qui, però, intendiamo operare comme il faut, facendo finta di essere i costruttori di uno strumento di misura e di trovarci impegnati nella sua calibrazione.

La calibrazione si fa, normalmente, *per confronto*, cioè attraverso lettura di un campione di unità di misura calibrato. Noi non disponiamo di un campione di tensione calibrato e il meglio che possiamo fare è produrre una d.d.p. (con generatore e partitore di tensione, come visto in precedenza), misurarla con il multimetro digitale, la cui calibrazione è nota, benché affetta da una incertezza sicuramente non trascurabile, e quindi usare la lettura del multimetro come campione (con la sua incertezza). In una forma leggermente modificata, dal punto di vista concettuale questo è quanto faremo nel metodo di calibrazione alternativa illustrato nella prossima sezione.

Basare la calibrazione su una singola misura implica, inevitabilmente, di assumere una *proporzionalità* diretta tra digit e V. Spulciando nelle specifiche del microcontroller si vede come questa affermazione possa essere non completamente corretta: infatti le specifiche suggeriscono che possa esserci un *offset* diverso da zero nella digitalizzazione. In altre parole, a una tensione di ingresso nulla entro le incertezze, $\Delta V = 0$, potrebbe corrispondere una lettura in digit diversa da zero, o viceversa. Inoltre la stessa dipendenza *lineare* tra digit e V deve, in linea di principio, essere verificata sperimentalmente, entro le incertezze della misura.

Per rispondere a queste esigenze occorre eseguire un esperimento in cui vengono raccolte le misure digitalizzate, qui indicate con il simbolo X, in corrispondenza di diversi valori della d.d.p. ΔV in ingresso. I dati possono quindi essere interpretati tramite un best-fit secondo la seguente funzione modello lineare, suggerita dalle specifiche di Arduino:

$$\Delta V = \alpha + \beta X \ , \tag{1}$$

dove i parametri α e β possono essere determinati tramite best-fit.

Nell'esempio da me realizzato, ho usato lo script `ardu2016.py` e lo sketch `ardu2016.ino` per costruire campioni di 256 misure (`nacqs = 1` nello script) acquisiti in corrispondenza di diversi valori ΔV_j realizzati ruotando in j-diverse posizioni l'alberino del potenziometro. Dalle indicazioni che lo script produce sulla console ho determinato il valore medio X_j della digitalizzazione effettuata. Come incertezze, per la misura con il multimetro ho seguito le consuete indicazioni del manuale, per i dati digitalizzati ho usato l'incertezza convenzionale ± 1 digit, a meno che la deviazione standard sperimentale del campione, indicata sempre sulla console, non fosse maggiore.

Il risultato delle misure di calibrazione è riportato in Fig. 6 assieme alla retta ottenuta dal best-fit, i cui risultati sono

$$\alpha = (19 \pm 3)\ \text{mV} \tag{2}$$
$$\beta = (4.87 \pm 0.02)\ \text{mV/digit} \tag{3}$$
$$\chi^2/\text{ndof} = 43/17 \tag{4}$$
$$\text{norm.cov.} = -0.65 \tag{5}$$
$$\texttt{absolute_sigma} = \texttt{False}\ ; \tag{6}$$

notate che nel best-fit si è considerata l'incertezza sui valori digitalizzati X_j attraverso la consueta procedura (errore equivalente determinato con propagazione e somma in quadratura). Come indicato dal grafico dei residui normalizzati, l'accordo tra dati e modello è scarso soprattutto per bassi valori di ΔV, dove probabilmente la risposta del digitalizzatore tende ad essere non lineare (siete

invitati a verificare di quanto l'accordo possa migliorare usando, per esempio, una dipendenza polinomiale di ordine superiore).

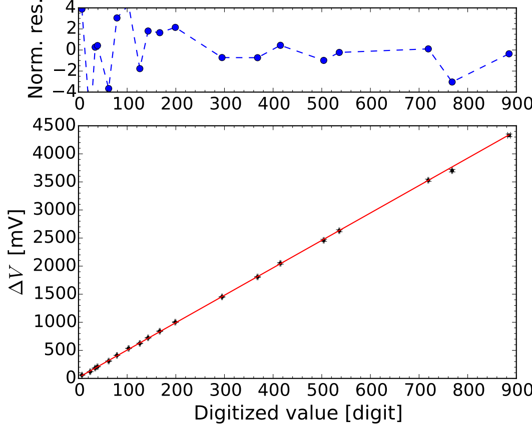

Figura 6. Risultato delle misure di calibrazione discusse nel testo: il pannello inferiore mostra dati e retta ottenuta dal best-fit, quello superiore il grafico dei residui normalizzati.

A. Calibrazione alternativa

La procedura di calibrazione alternativa (così chiamata per nostra convenzione) sfrutta una relazione di proporzionalità diretta tra d.d.p. in V e valore digitalizzato X, secondo la

$$\Delta V = \xi X \ . \tag{7}$$

Dunque in questa relazione la presenza dell'offset (rappresentato dal parametro α sopra determinato) è trascurata.

Evidentemente, se fosse nota la tensione di riferimento V_{ref}, che corrisponde alla massima lettura possibile, dunque a $X = 1023$ digit, sarebbe possibile ricavare ξ dalla semplice relazione

$$\xi = \frac{V_{ref}}{1023} \ . \tag{8}$$

Il valore di V_{ref} è però noto (in forma nominale) solo usando il riferimento interno da 1.1 V. Negli altri casi, si può utilizzare un'utile scorciatoia suggerita dal tanto materiale disponibile in rete. Infatti V_{ref} è attesa corrispondere anche alla massima tensione che può essere prodotta da Arduino alle sue porte di uscita. Quindi misurando con il multimetro digitale questa tensione è possibile conoscere in maniera immediata ξ e ottenere una calibrazione (approssimativa) dello strumento.

Nello sketch qui utilizzato, come abbiamo già sottolineato, Arduino viene istruito a usare la porta corrispondente al pin 7 (collegato con filo arancione/rosso a una boccola volante rossa) come uscita digitale e a porla a "livello alto", cioè a tenerla "accesa" per tutta la durata dell'esperienza. La nostra ipotesi, supportata da molte informazioni disponibili in rete, è che la tensione misurata fra questa porta e la linea di terra in queste condizioni, ΔV_{pin7}, sia pari (approssimativamente) a V_{ref}. Nell'esempio qui riportato, si è ottenuto $\Delta V_{pin7} = (4.94 \pm 0.03)$ V, a cui corrisponde $\xi = (4.83 \pm 0.03)$ mV/digit: questo valore è compatibile con quello del parametro β prima determinato con il best-fit, circostanza che fornisce una prima indicazione qualitativa sulla validità della procedura di calibrazione alternativa (almeno per tutte le situazioni in cui si ritiene non necessario spingersi ai massimi livelli possibili di accuratezza di calibrazione).

Una valutazione un po' più quantitativa può essere ottenuta come descritto qui nel seguito. Dal best-fit lineare della calibrazione è possibile determinare il valore *previsto* ΔV_{prev} corrispondente a una arbitraria lettura digitalizzata X (ΔV_{prev} è quindi da intendersi come una funzione di X). In questa previsione è utile tenere conto della *covarianza* dei due parametri di fit, secondo quanto già conosciamo; ricordiamo infatti che vale la relazione

$$\delta V_{prev} = \sqrt{C_{11}^2 + C_{22}X^2 + 2C_{12}X} \ , \tag{9}$$

dove δV_{prev} è l'incertezza della previsione, C_{ij} sono gli elementi della *matrice di covarianza* ottenuti dal best-fit lineare (numerico), X è il valore della lettura digitalizzata. La stessa operazione possiamo farla per la previsione basata sulla calibrazione alternativa, $\Delta V_{prev,alt} = \xi X$: stavolta l'impiego di un unico parametro rende inutile preoccuparsi della covarianza (che non è definita in questo caso), per cui, con ovvio significato dei simboli, possiamo porre $\delta V_{prev,alt} = \Delta\xi X$.

A questo punto possiamo costruire le cosiddette "curve di confidenza" (nome convenzionale), cioè le funzioni $\Delta V_{prev} \pm \delta V_{prev}$ e $\Delta V_{prev,alt} \pm \delta V_{prev,alt}$; il risultato è mostrato in Fig. 7, dove si è impiegata la rappresentazione logaritmica per mettere meglio in evidenza le piccole differenze in valore assoluto (per questo grafico, X è stata costruita con un array equispaziato logaritmicamente). Se i due metodi di calibrazione portassero a risultati in accordo fra loro entro le incertezze, le "bande di confidenza" dovrebbero essere una dentro l'altra. Apparentemente (cioè con la risoluzione consentita dal grafico) questo non si verifica sempre e, in particolare, per bassi valori di ΔV, ovvero di X, ci sono discrepanze facilmente apprezzabili. Infatti è ovvio che, pur se l'intercetta del modello lineare (parametro α del fit) è piccola in valore assoluto, essa produce degli effetti nella calibrazione di valori digitalizzati corrispondenti a pochi digit.

Allo scopo di ottenere un'indicazione quantitativa, è utile creare la funzione ∂, simbolo con cui indichiamo la *massima* discrepanza (in valore assoluto) tra i risultati delle due calibrazioni, normalizzata rispetto alla lettura ΔV. Dal punto di vista matematico, si può porre, per esempio,

$$\partial = \frac{max\{|\Delta V_{prev} - \Delta V_{prev,alt}|\}}{\Delta V_{prev}} \ . \tag{10}$$

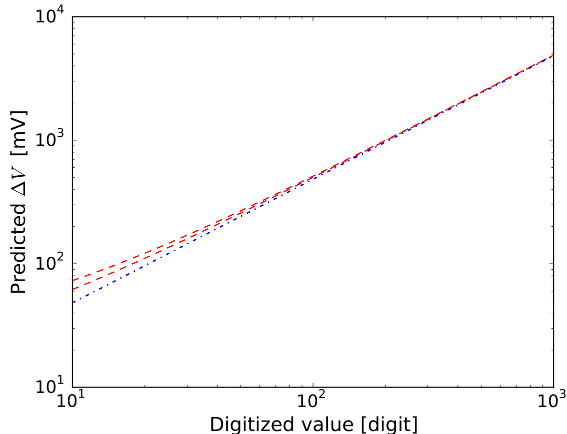

Figura 7. "Curve di confidenza" per i valori ΔV previsti in funzione della lettura digitalizzata X: secondo quanto discusso nel testo, le linee tratteggiate rosse rappresentano il risultato ottenuto considerando la calibrazione con best-fit lineare, quelle punto-linea blu la calibrazione alternativa.

Il grafico di questa grandezza è rappresentato in Fig. 8: si vede come la discrepanza relativa ∂ tra i due metodi di calibrazione sia tutt'altro che trascurabile per bassi valori digitalizzati, ma anche come essa scenda sotto il 5% quando la lettura del digitalizzatore è superiore al centinaio.

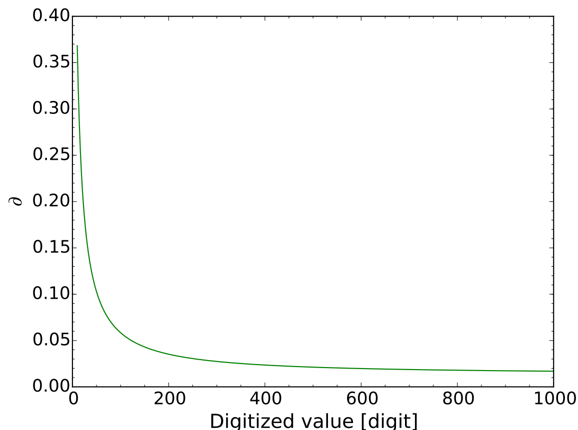

Figura 8. Discrepanza relativa ∂ tra i due metodi di calibrazione, come definita nel testo, in funzione del valore digitalizzato.

VIII. QUALCHE APPARENTE STRANEZZA

Da ultimo, in questa sezione si riporta un'osservazione che può essere facilmente registrata in modo involontario, ma che qui è costruita apposta. Può succedere talvolta che si esegua una digitalizzazione mantenendo scollegato l'ingresso di Arduino (ovvero collegato su un carico resistivo elevato: si ottengono risultati simili a quelli qui mostrati anche chiudendo l'ingresso di Arduino, il pin `A0`, su una resistenza dell'ordine del Mohm). In queste condizioni ci si potrebbe aspettare di ottenere una lettura costantemente nulla, a parte piccole fluttuazioni.

Un risultato esempio è mostrato in Fig. 9 (la figura è costruita come Fig. 3 e quindi riporta anche l'istogramma delle occorrenze): si vede che le letture sono ben diverse da zero e che la loro distribuzione popola diversi bins (non solo quello corrispondente alla lettura nulla).

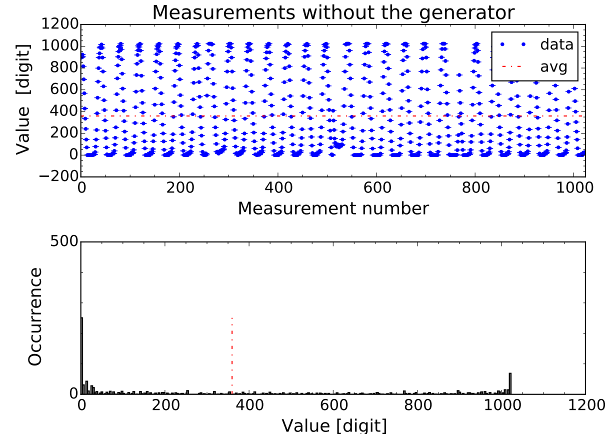

Figura 9. Analogo di Fig. 3 dove, però, il generatore è stato scollegato, lasciando "volante" la boccola collegata al pin `A0` di Arduino, ovvero *flottante* la d.d.p. in ingresso al digitalizzatore.

L'interpretazione di questa apparente stranezza è piuttosto immediata: quelle registrate sono letture dovute a "disturbi", che vengono raccolti (in maniera evidentemente efficace) dall'ingresso del digitalizzatore. Poiché le digitalizzazioni sono pressoché equispaziate nel tempo, il grafico suggerisce un andamento periodico, o quasi-periodico, dei disturbi e un'analisi che tiene conto anche del tempo di digitalizzazione indica in circa 50 Hz la frequenza (prevalente) dei disturbi. Allora è evidente che essi hanno origine nella corrente di rete, che, essendo alternata, dà luogo, attraverso meccanismi che vi saranno chiari proseguendo nel corso, a una sorta di d.d.p., anche alternata e alla stessa frequenza di 50 Hz (o un suo multiplo). In futuro indicheremo talvolta fenomeni di accoppiamento dei disturbi agli strumenti di misura come *rumori di pick-up*.

Della presenza di queste fluttuazioni, però, non ci si rende generalmente conto usando altri strumenti di misura: lasciando scollegato il multimetro digitale configurato come voltmetro (in corrente continua), non si osservano variazioni della lettura che possano essere ascritte a questo tipo di disturbi. Ci sono diversi motivi che possono spiegare lo specifico comportamento di Arduino confrontato con quello del voltmetro digitale: quest'ultimo, infatti, è costruito per leggere tensioni continue, ha

un tempo di refresh del display piuttosto lungo (sicuramente più lungo degli intervalli di campionamento qui impiegati) e, in pratica, esso media a zero, ovvero "filtra", i disturbi alternati. In secondo luogo, il layout del multimetro digitale è certamente più accurato di quello della scheda Arduino, almeno come la usiamo noi, per cui i disturbi alternati potrebbero essere "schermati" e attenuati in ampiezza. Inoltre il multimetro è alimentato a batteria e non ha di per sé alcun collegamento fisico con la rete elettrica, o la linea di terra. Infine, la resistenza di ingresso di Arduino potrebbe essere superiore rispetto a quella (già molto alta) del multimetro usato come voltmetro, e questo potrebbe aumentare l'ampiezza della d.d.p. creata dal disturbo.

In ogni caso, ricordate sempre che non collegare nulla a uno strumento di misura non significa necessariamente porre pari a zero la grandezza in ingresso: azzerare il segnale letto significa infatti collegare l'ingresso alla linea di massa, o di terra.

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 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 nominalmente 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, prevede 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 eseguire misure *sincrone*: allo scopo, verranno implementate strategie in grado di "triggerare" (brutto termine che vi diventerà molto familiare) l'acquisizione in contemporanea con il verificarsi di determinati eventi.

[4] Il motivo è banale. Il microcontroller di Arduino, come qualsiasi CPU, ha ritardi, o tempi di latenza, durante l'esecuzione di un ciclo di programma, che non possono essere determinati a priori. Infatti, oltre ad eseguire il programma, la CPU 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 interpretabile come statistica.

[5] Il tempo minimo di campionamento non è un parametro direttamente noto dai datasheet del microcontroller. Secondo diverse fonti, esso è dell'ordine delle (poche) decine di microsecondi. L'esperienza discussa in questa nota suggerisce, come sarà chiarito in seguito, che la digitalizzazione avviene all'interno di una finestra temporale inferiore a 20 μs, almeno per la scheda Arduino impiegata in questo esempio.

[6] La scelta di R_1 determina, secondo le regole dei partitori di tensione, il massimo valore della d.d.p. ΔV che può essere prodotta.

[7] 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.

[8] 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.

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

[10] In futuro torneremo su questi "disturbi", indicandoli talvolta come *rumore*. Molto grossolanamente, infatti, chiameremo rumore tutte le fluttuazioni dei valori letti diverse da cause fisiche che possiamo controllare.