Arduino e campionamento/digitalizzazione di segnali continui

francesco.fuso@unipi.it

(Dated: version 7 - FF, 24 ottobre 2019)

Queste nota ha lo scopo principale di illustrare le modalità di funzionamento di base tipiche delle esercitazioni pratiche in cui si fa uso di Arduino [1], prendendo spunto da quella finalizzata a esaminarne il comportamento in misure di d.d.p. continue e determinarne la calibrazione. Essendo la prima occasione di impiego di Arduino, adeguato spazio è dato a illustrare alcuni dettagli pratici.

I. ANALOGICO VS DIGITALE

Convenzionalmente è possibile distinguere tra misure *analogiche*, per esempio quelle basate sullo spostamento idealmente continuo di una lancetta su un quadrante, e *digitali*, quelle che forniscono un *numero intero* a cui, attraverso opportune calibrazioni, si attribuisce un valore fisico eventualmente in virgola mobile. Queste tipologie di misura possono essere applicate sia a grandezze discrete, "naturalmente" digitali, che a grandezze "non discrete", "naturalmente" analogiche, quali generalmente tutte le grandezze fisiche di interesse, d.d.p., intensità di corrente, resistenze, misurate nel nostro corso [2].

Dal punto di vista pratico anche una qualsiasi misura analogica può dare luogo a un risultato intero (opportunamente dimensionato), per esempio quando essa, codificata in modo opportuno (per esempio scegliendo l'unità di misura in modo che non ci siano cifre significative decimali o eseguendo un'approssimazione, o troncamento) viene trascritta su un foglio, o inserita in un computer per il successivo trattamento. Caratteristica specifica della misura digitale è quella di non richiedere alcun passaggio ulteriore per essere ricondotta a un numero intero.

A. Digitalizzazione e campionamento: digitalizzatore modello

I meccanismi di digitalizzazione, di cui daremo un breve esempio concettuale in seguito, sono inerentemente accompagnati da processi di *campionamento*. Campionare significa prendere un pezzettino, in senso generalmente temporale, della grandezza da misurare, che quindi viene analizzata solo per questo pezzettino. In pratica, la digitalizzazione avviene in un intervallo temporale finito e non nullo, cioè essa richiede del tempo per essere portata a termine, per cui è inevitabile che la conversione da analogico a digitale avvenga per pezzettini del segnale sotto analisi, di durata non nulla. Dunque digitalizzazione e campionamento sono concetti intimamente legati l'un l'altro.

Un dispositivo che consente la digitalizzazione attraverso campionamento si chiama in genere *convertitore analogico-digitale* (ADC - Analog to Digital Converter, o convertitore A/D). Nelle nostre esercitazioni pratiche l'ADC è contenuto nella scheda Arduino, o anche, ovviamente, nel multimetro digitale. Questi strumenti hanno principi di operazione che non conosciamo nei dettagli, ma ugualmente, a scopo prevalentemente didattico, possiamo descrivere il funzionamento di un digitalizzatore *modello*. I suoi ingredienti fondamentali sono:

1. un *clock*, ovvero un dispositivo in grado di generare impulsi, di durata virtualmente trascurabile, equispaziati nel tempo, così come fa un orologio;

2. un *contatore*, cioè un dispositivo in grado di contare gli impulsi di cui sopra, dotato di un circuito di reset (di azzeramento) opportunamente comandabile;

3. un'*interfaccia digitale*, cioè, per intenderci, un processore in grado di trattare (registrare, visualizzare, etc.) il conteggio prodotto dal contatore;

4. un *generatore di rampa*, cioè un dispositivo, opportunamente triggerabile, cioè con partenza comandabile dall'esterno, in grado di fornire una d.d.p. crescente linearmente con il tempo;

5. un *comparatore*, cioè un dispositivo in grado di comparare i livelli (le d.d.p., riferite ovviamente alla stessa linea di massa) di due ingressi e fornire in uscita un segnale dipendente dall'esito della comparazione (per intenderci, zero o uno a seconda che prevalga uno o l'altro dei due segnali in ingresso).

La Fig. 1 illustra lo schema a blocchi, semplificato, del sistema [pannello (a)] e mostra uno schemino della temporizzazione (timing) del circuito, ovvero della sequenza temporale delle operazioni da esso compiute [pannello (b)].

Per capire il funzionamento di questo digitalizzatore modello cominciamo subito con il notare che il conteggio degli impulsi di clock è un'operazione inerentemente digitale, dato che il suo esito è sicuramente un numero intero. Infatti, almeno in linea di principio, ogni operazione di misura di intervalli temporali può essere ricondotta al conteggio dei cicli di oscillazione di un orologio. Agli ingressi del comparatore sono inviati il segnale (d.d.p.) da misurare e l'uscita (d.d.p.) del generatore di rampa. Supponiamo poi che la partenza della rampa, cioè il trigger del generatore che produce la rampa, scatti contemporaneamente al reset del contatore del tempo: dunque il contatore segna zero all'inizio del processo, quando il valore della rampa è pure zero (o altro livello noto). Infine,

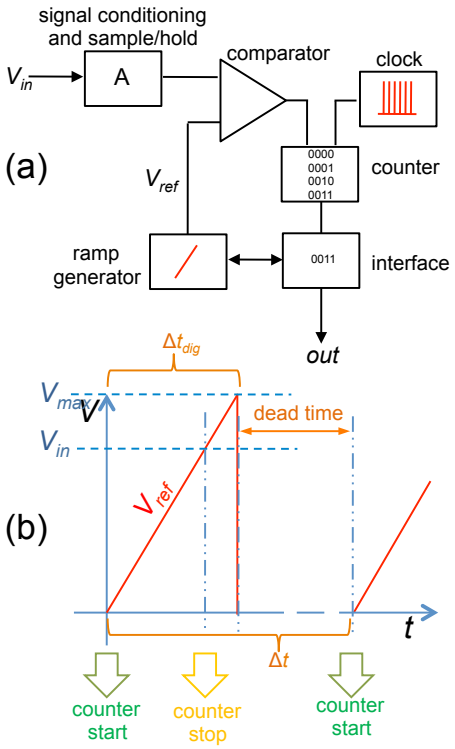

Figura 1. Schema a blocchi semplificato (a) e schema del timing di funzionamento (b) di un digitalizzatore modello; V_{sig} e V_{ramp} indicano rispettivamente la d.d.p. del segnale (incognita da misurare) e la d.d.p. prodotta dal generatore di rampa, che aumenta linearmente nel tempo. Lo schema a blocchi non riporta esplicitamente, per semplicità tipografica, lo stadio di sample-and-hold brevemente citato nel testo.

supponiamo che l'uscita del comparatore vada a comandare l'interfaccia digitale, cioè sia in grado di "bloccare" il conteggio. Il contatore conta (1, 2, 3,..) fin quando il segnale incognito in ingresso è inferiore alla rampa. Quando invece la rampa supera il segnale incognito, il processo si interrompe e l'interfaccia digitale acquisisce ("blocca") il conteggio. Grazie alla linearità della rampa con il tempo, questo conteggio è proporzionale al valore del segnale in ingresso, che quindi risulta digitalizzato, cioè convertito in un numero intero. Un'opportuna calibrazione consente di ricondurre il conteggio al valore di d.d.p. (in unità fisiche) del segnale incognito entro una determinata incertezza di calibrazione. Inoltre è ovvio che, nelle implementazioni pratiche, il segnale incognito deve essere sottoposto a qualche forma di "condizionamento" prima di arrivare all'ingresso del comparatore, cioè deve essere opportunamente amplificato o attenuato, eventualmente cambiato di segno e traslato rispetto allo zero. Anche il circuito di condizionamento soffre di incertezze di calibrazione, delle quali tenere conto nella determinazione dell'errore complessivo di misura.

Poiché la conversione da analogico a digitale è, di fatto, una misura di tempo, è ovvio che essa richiede del tempo per essere completata. Dunque un digitalizzatore non può operare continuamente, per cui viene confermata l'affermazione precedente sulla necessità che il segnale digitalizzato sia sempre e inevitabilmente campionato in un (generalmente piccolo) pezzettino di tempo.

In questo intervallo di tempo sarebbe opportuno che la d.d.p. incognita all'ingresso del comparatore risultasse costante e, dato che questo non si verifica necessariamente (non si verifica affatto nella maggior parte delle situazioni sperimentali di interesse), è necessario applicare un'ulteriore forma di condizionamento. Questo condizionamento viene normalmente attuato da un circuito, detto *sample-and-hold*, che mantiene costante, al valore che ha all'inizio della digitalizzazione, il segnale in ingresso al comparatore per tutta la durata della digitalizzazione stessa. Un semplice modello di sample-and-hold è un condensatore che viene caricato in maniera praticamente istantanea, mantenendosi a un potenziale costante per la durata della digitalizzazione. Al termine, cioè quando il contatore del tempo viene bloccato, il condensatore deve essere scaricato, in maniera altrettanto istantanea, per permettere una nuova digitalizzazione, e quindi occorre un opportuno circuito di controllo in grado di compiere tali operazioni in maniera sincrona con l'operazione del digitalizzatore [3].

La descrizione del meccanismo di digitalizzazione mette in evidenza immediatamente quali siano le due principali figure di merito di un digitalizzatore:

- il massimo rate di campionamento, che è evidentemente dipendente dalla rapidità di risposta del comparatore e dalla ripidità della rampa prodotta; esso è normalmente espresso in Sa/s (samples per secondo) e qualche volta in Hz (eventi di campionamento per secondo);

- la dinamica, o profondità di digitalizzazione, cioè il massimo numero di conteggi che può essere registrato (ovviamente nel tempo necessario alla misura), normalmente espresso in *bit*, dato che nel mondo digitale vige la codifica binaria.

Entrambe queste figure di merito hanno ripercussioni sulla qualità della misura. Il massimo rate di campionamento influenza direttamente la possibilità di seguire le variazioni di segnali rapidamente dipendenti dal tempo. La dinamica, o profondità di digitalizzazione, ha a che fare con la portata e la sensibilità della misura. Inoltre è evidente che, a prescindere da ogni altra eventuale causa di errore legata alla calibrazione, esiste un'*incertezza di digitalizzazione* che vale (almeno e salvo ulteriori considerazioni) un bit. Infatti la capacità dinamica finita implica che il comparatore scatti quando il segnale si trova all'interno di un intervallo la cui ampiezza è pari a quella di un singolo bit (ovvero *un digit* nel nostro linguaggio comune), per cui il valore del segnale può essere determinato solo all'interno di questo intervallo.

II. CARATTERISTICHE PRINCIPALI DI ARDUINO

Come ben sapete, in laboratorio facciamo uso di Arduino per diversi scopi sperimentali, il principale dei quali è quello di digitalizzare delle d.d.p. generalmente dipendenti dal tempo usando le porte analogiche disponibili (6 ingressi distinti, marcati `A0 - A5`). La dinamica di digitalizzazione, che vale 10 bit (cioè 1024 livelli distinti, da 0 a 1023), è un dato ben noto. Il massimo rate di campionamento, invece, non è riportato in maniera chiara nei datasheets, dove è indicato un ampio intervallo, evidentemente dipendente dalle condizioni di operazione, che mostra come il tempo minimo necessario per concludere una singola digitalizzazione, qui chiamato Δt_{dig}, sia di almeno 13 μs. Prove sperimentali del tipo di quelle descritte nel seguito di questa nota mostrano che $\Delta t_{dig} \simeq 13 - 20$ μs.

Tuttavia, l'operazione intrinseca di digitalizzazione non è la sola coinvolta nel processo di conversione da analogico a digitale. In altre parole, due conversioni analogico digitale consecutive non possono essere eseguite se intervallate da un tempo pari solo a Δt_{dig}, poiché del tempo addizionale è necessario per completare tutte le operazioni coinvolte. In questa nota faremo riferimento al simbolo Δt per indicare l'intervallo di tempo *effettivo* tra una conversione e la successiva (Δt_{nom} indicherà il valore *nominale* di tale intervallo temporale). Per quanto sopra affermato, è $\Delta t > \Delta t_{dig}$; al minimo, Arduino funziona stabilmente e senza casini evidenti se $\Delta t \geq 40 - 50$ μs (approssimativamente). Il massimo rate di campionamento è quindi di circa 20 kSa/s; di conseguenza usando Arduino è difficile seguire segnali che variano in maniera significativa su scale temporali inferiori a diverse decine di microsecondi. Naturalmente esistono digitalizzatori ben più performanti, e, per esempio, molti oscilloscopi digitali permettono di operare a 1 GSa/s, o anche oltre.

Dato che qui siamo a riassumere le principali caratteristiche di Arduino, ricordiamone anche delle altre, cominciando con quelle di carattere elettrico. La resistenza di ingresso del digitalizzatore, come dichiarata nei datasheets, è molto alta (100 Mohm nominali) per segnali stazionari (continui). Oltre agli ingressi analogici, Arduino ha poi la possibilità di controllare ben 14 porte digitali, configurabili come input o output. Queste porte possono assumere o leggere un valore binario (zero o uno) a seconda che siano a livello "basso" (ovvero, in rappresentazione binaria, 0, in unità fisiche circa 0 V) o "alto" (ovvero 1, tipicamente pari alla tensione di riferimento del digitalizzatore, cioè $V_{ref} \simeq 5$ V)[4]. Infatti il significato dell'aggettivo digitale ad esse attribuito significa proprio che hanno un comportamento binario (se usate come output, sono accese o spente, se utilizzate come input stabiliscono che in ingresso c'è un segnale acceso o spento). Configurate come uscite, queste porte sono utili per "controllare" l'esperimento, come per esempio nella carica/scarica del condensatore. La massima corrente che può essere ottenuta è, nominalmente, di 20 mA (50 mA in totale se se ne usa più di una). Naturalmente accensione e spegnimento delle porte non sono operazioni realmente istantanee, cioè la commutazione tra livello alto e basso non può che avvenire in un tempo finito: si può verificare facilmente, con una semplice misura all'oscilloscopio, che soprattutto la fase di spegnimento segue un andamento esponenziale, con un tempo caratteristico di alcune decine di microsecondi (paragonabile al tempo minimo di campionamento, che quindi stabilisce una sorta di "limite di velocità" per Arduino).

Sei tra queste porte digitali, quelle marcate con un tilde nelle serigrafie, possono operare come output in una modalità detta `PWM` (*Pulse-Width Modulation*). In questa modalità le porte generano treni di impulsi a frequenza fissa (e piuttosto bassa, un po' meno di 1 kHz) con duty-cycle regolabile via software (può essere aggiustato su 8 bit, cioè 256 distinti valori) [5]. Di questa tipologia di porte ci serviremo in una futura esercitazione per realizzare una sorta di convertitore digitale/analogico (DAC, in gergo).

Infine, avremo sicuramente modo di usare alcune di queste porte come ingressi digitali, in particolare per scopi di sincronizzazione dell'acquisizione con specifici eventi. Di tutto questo ci occuperemo eventualmente in sede di presentazione e discussione delle singole esperienze.

III. MISURA DI D.D.P. CONTINUE

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, eventualmente della deviazione standard e realizzazione di istogrammi delle occorrenze, etc.), oltre a permettere di eseguire una calibrazione del digitalizzatore tramite misura di diversi valori e confronto con la lettura del multimetro digitale, usato qui come riferimento.

Per ottenere gli scopi della presente esercitazione pratica 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` - ovviamente la d.d.p. è riferita a un altro ingresso di Arduino, quello corrispondente alla linea di massa, o terra, secondo quanto specificato in seguito). 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 Ar-

duino prevede che esso immagazzini temporaneamente i risultati delle misure nella sua memoria interna, per poi trasferire al computer il *record* contenente tutti i dati, in "un colpo solo" al termine dell'acquisizione.

A. Configurazione di misura

L'esercitazione pratica prevede di eseguire in maniera automatica molte misure (centinaia o migliaia) della stessa d.d.p., qui chiamata ΔV. Tale grandezza viene letta in forma di numeri interi compresi tra 0 e 1023. Le tante misure vengono acquisite in successione, dunque a istanti diversi. In questa esperienza l'acquisizione è *asincrona*, cioè non deve partire in contemporanea con qualche ben definito evento esterno, e *non interessa* conoscere l'istante in cui avviene l'acquisizione, poiché non abbiamo necessità di studiare l'andamento temporale della grandezza considerata, che è supposta continua. Tuttavia, come preparazione a ulteriori esperienze con Arduino e anche allo scopo di studiare l'incertezza nella misura dei tempi, l'esperimento 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 *nominalmente* equispaziati per un tempo Δt_{nom}: l'analisi dei dati corrispondenti permetterà di verificare entro quale accuratezza tale equispaziatura sia effettivamente realizzata. L'origine di tale incertezza è molteplice: ad essa contribuisce la misura del tempo che Arduino è in grado di compiere (il suo orologio interno si basa su un oscillatore al quarzo la cui frequenza di risonanza è 16.000 MHz, dunque con tolleranza di circa 63 ppm), ma probabilmente il meccanismo principale è l'aleatorietà (presenza di latenze o tempi morti) con cui il microcontroller che si trova all'interno di Arduino compie le proprie operazioni logiche fondamentali. Questa situazione non è troppo diversa da quella che si incontra quotidianamente con telefonini, PC, o altro nel momento in cui si riscontra una qualche "mancanza di fluidità" nell'esecuzione dei comandi.

B. Partitore con potenziometro

Poiché è di interesse 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 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 dotato di 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 circuitale 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. In laboratorio sono disponibili diversi potenziometri, la maggior parte dei quali ha $R_V = 4.7$ kohm o $R_V = 470$ 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 (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 e dunque "perturba" in modo trascurabile il circuito. Nel mio esempio, dove ho impiegato il potenziometro con $R_V = 4.7$ kohm, 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 a pettine 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

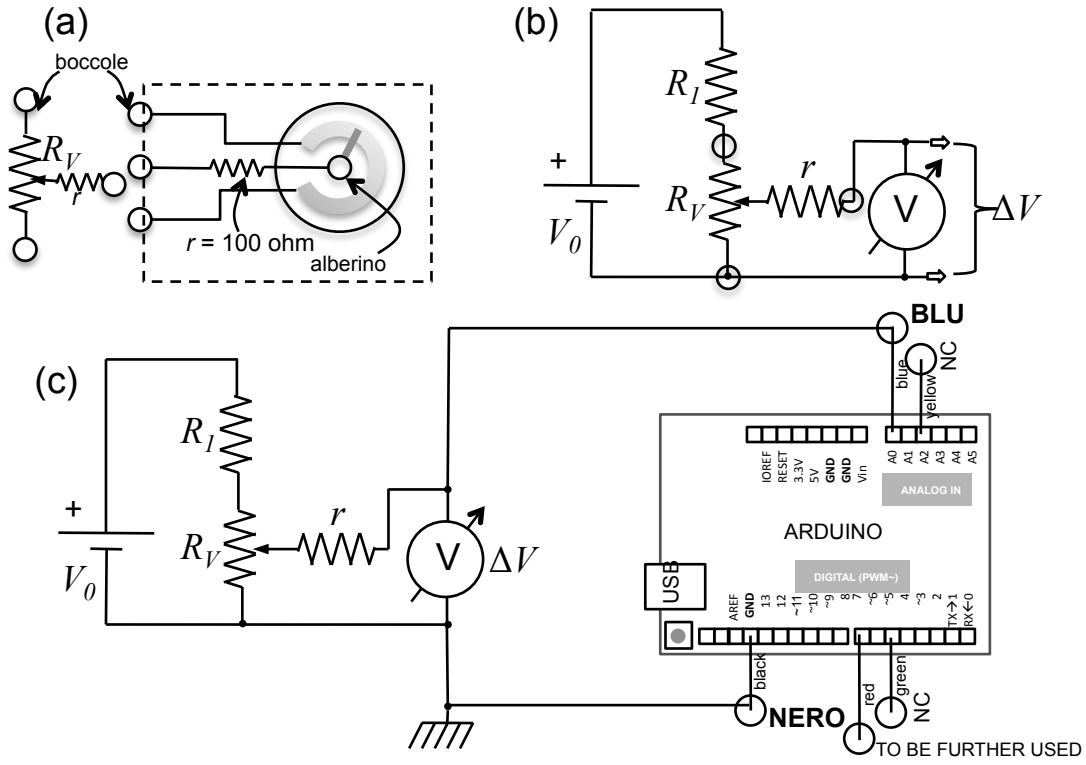

Figura 2. Rappresentazione schematica e costruttiva del potenziometro "visto da sotto" (a), schema del partitore di tensione (b), configurazione del circuito di misura comprendente Arduino (c). Nel pannello (a) è riportato uno dei simboli con cui si indica un potenziometro negli schemi elettronici. Nel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

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`. L'indicazione è un'abbreviazione di ground, cioè terra: infatti questa boccola è collegata alle linee di riferimento che Arduino usa come potenziale nullo (*massa*) che, attraverso la connessione USB, sono connesse alle linee di massa del computer e di lì, tramite il connettore di alimentazione della rete elettrica, alla terra dell'impianto di distribuzione elettrica. 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 eterno.

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 impiegare 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 non volatile 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 generale collegate Arduino al resto del circuito *solo dopo aver effettuato l'upload dello sketch*.

Ricordiamo infatti che Arduino viene istruito a compiere determinate operazioni attraverso un semplice programma, scritto in un file di testo (con estensione `.ino`) da trasferire (uploadare) nel microcontroller attraverso il programma Arduino, o Arduino IDE, presente nei computer di laboratorio. La comunicazione con il computer e la gestione dell'esperimento vengono invece controllate da Python usando appositi script.

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. Tenete sempre presente che nei computer di laboratorio la libreria `serial` può essere caricata solo lanciando Python da terminale (non da Pyzo).

A causa delle piccole dimensioni della memoria SRAM di Arduino, solo 2 kB, il numero di misure distinte che possono essere eseguite e registrate 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 [7]. 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 9600 baud (1 baud = 1 bit/s), non molto elevata ma sicuramente adeguata agli scopi (in qualche caso potrebbe essere stata aumentata, cioè moltiplicata per un multiplo di 2), 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 esercitazione pratica*, 100 μs, per cui il "5" significa che i dati saranno campionati con intervalli nominali di $\Delta t_{nom} = 5 \times 100$ μs $= 500$ μs. 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 su Δt, ovvero dell'*effettivo* intervallo temporale tra le operazioni di digitalizzazione.

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 il record venga trasferito al computer quando effettivamente pronto. 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: 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 coppia di dati viene aggiunta al file di testo prodotto dallo script. 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'esercitazione pratica 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 sulla console 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 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 esso sia suddiviso in diverse parti, che nella nostra implementazione sono tre 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 assu-

mere valori negativi) occupa 2 byte (1 byte equivale a 8 bit), cioè due caratteri, e permette quindi di individuare $2^{8+8} = 2^{16}$ valori interi differenti; un intero `long` richiede invece 4 byte.

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 9600 baud (`Serial.begin(9600);`), o altra velocità, sempre coerente con quanto dichiarato nello scritp di Python, 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 decine di μs. Senza entrare troppo nei dettagli, lo scopo delle istruzioni 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'esercitazione trattata in questa nota la 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 aspetti 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, in particolare nella fase di trasferimento dati via porta seriale USB (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 [8]. Attraverso quanto descritto in seguito verificheremo la corrispondenza tra valore effettivo e nominale di tale intervallo.

Terminata l'acquisizione dei 256 punti sperimentali, ovvero la costruzione del record composto da 256 coppie di dati (d.d.p. in digit e tempo in μs) comincia il ciclo di scrittura 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. CALIBRAZIONE DI ARDUINO

Nelle nostre esercitazioni pratiche Arduino viene impiegato principalmente come misuratore di d.d.p.: anche se spesso sarà sufficiente per i nostri scopi conoscere le tensioni misurate in unità arbitrarie di digitalizzazione (digit), talvolta 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_{max} [vedi Fig. 1(a)] che rappresenta il valore massimo della d.d.p. (in unità fisiche) che può essere campionata e quindi corrisponde alla lettura di 1023 digit. In condizioni di operazione ordinarie (senza cioè scegliere il riferimento interno da 1.1 V nominali), V_{max} dipende dall'alimentazione che Arduino riceve tramite USB, tipicamente attorno a 5 V. 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. 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 con la sua incertezza come campione di riferimento. Inoltre, per verificare la calibrazione su un

intervallo ampio di valori, eseguiamo 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. Notate che la funzione modello contiene un termine di *offset* costante (α), previsto per tenere conto della circostanza che un ingresso "nullo" risulti in una lettura "non nulla", e viceversa.

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. Osservate che, a rigore, la validità di questa procedura di attribuzione dell'incertezza è opinabile, poiché, ad esempio, sappiamo che l'incertezza è generalmente una sovrastima della deviazione standard. Inoltre, definendo deviazione standard il risultato sperimentale, stiamo implicitamente supponendo che le misure abbiano una qualche distribuzione, affermazione sicuramente ragionevole, ma non verificata rigorosamente.

I dati delle misure di calibrazione sono rappresentati in Fig. 3 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 dell'errore efficace. 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).

A. Calibrazione alternativa

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

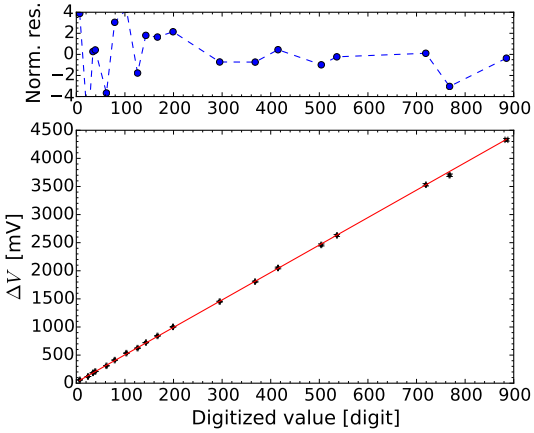

Figura 3. Risultato delle misure di calibrazione discusse nel testo: il pannello inferiore mostra dati e retta ottenuta dal best-fit, quello superiore il grafico dei residui normalizzati.

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

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_{max}, che corrisponde al massimo valore di d.d.p. che può essere digitalizzata e dunque a $X = 1023$ digit, sarebbe possibile ricavare ξ dalla semplice relazione

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

Il valore di V_{max} 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_{max} è attesa corrispondere anche alla massima tensione che può essere prodotta da Arduino sulle sue porte digitali di uscita. Quindi misurando con il multimetro digitale questa tensione è possibile conoscere in maniera immediata ξ e ottenere una calibrazione (approssimativa, ma spesso ragionevole per i nostri scopi) dello strumento.

Nello sketch utilizzato, come 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. Sulla base delle informazioni disponibili in rete, assumiamo che la tensione misurata fra questa porta e la linea di terra in queste condizioni, ΔV_{pin7}, sia pari nominalmente a V_{max}. 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.

Una valutazione semi-quantitativa dell'accordo tra le due procedure di calibrazione 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 occrre tenere conto della *covarianza* dei due parametri di fit, secondo quanto già conosciamo; ricordiamo infatti che vale la relazione

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

dove δV_{prev} è l'incertezza della previsione, C_{ij} sono gli elementi della *matrice di covarianza* ottenuti dal best-fit, 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, 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. 4, 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.

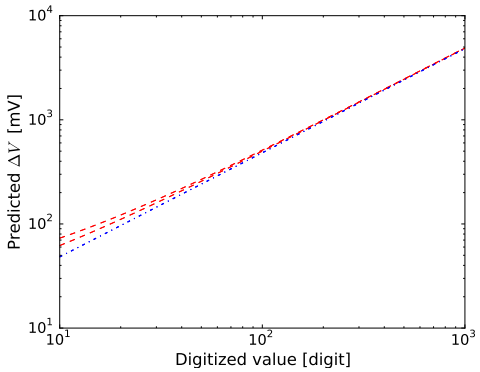

Figura 4. "Curve di confidenza" per i valori ΔV previsti in funzione della lettura digitalizzata X: secondo quanto discusso nel testo, le linee tratteggiate rosse rappresentano il risultato ottenuto considerando la calibrazione con best-fit lineare, quelle punto-linea blu la calibrazione alternativa.

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,

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

Il grafico di questa grandezza è rappresentato in Fig. 5: 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.

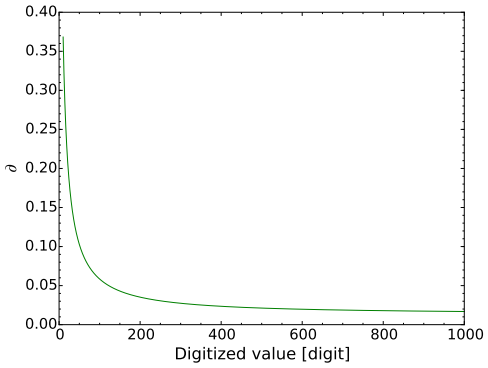

Figura 5. Discrepanza relativa ∂ tra i due metodi di calibrazione, come definita nel testo, in funzione del valore digitalizzato.

VII. ANALISI STATISTICA DI DATI ESEMPIO

L'esperimento trattato in questa nota consente 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 stata 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, che purtroppo resterà parzialmente frustrato, 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.

A. Campione di misure digitalizzate

La Fig. 6 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.

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).

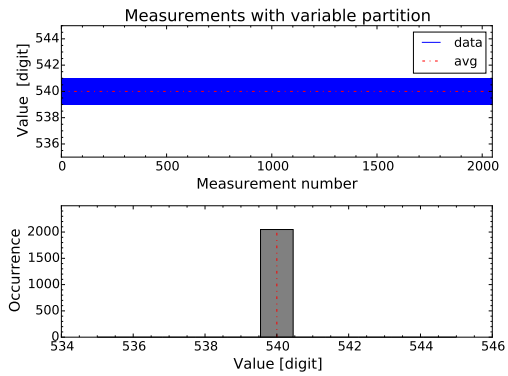

Figura 6. Esempio di analisi del campione di dati acquisito per $\Delta V = (2.65 \pm 0.02)$ V, misurata con il multimetro digitale. Il pannello superiore riporta il grafico delle misure, a cui è stata attribuita un'incertezza convenzionale ±1 digit, il pannello inferiore riporta l'istogramma delle occorrenze. Il valore medio digitalizzato è 540 digit e la deviazione standard sperimentale è nulla, secondo quanto descritto nel testo.

La situazione considerata, in cui si registra una deviazione standard sperimentale nulla, è tutt'altro che patologica. Anche supponendo l'esistenza di una distribuzione dei valori, per esempio normale, l'accuratezza di 1 digit con la quale il valore può essere registrato è evidentemente troppo scarsa per ricostruire la distribuzione e dunque apprezzarne la deviazione standard. Poiché però nessuna misura può avere incertezza nulla, scegliamo convenzionalmente di individuare un'incertezza *arbitraria*, che altrettanto convenzionalmente poniamo pari a ±1 digit. Notate che, anche se presumibilmente questa incertezza sovrastima l'effettiva deviazione standard, possiamo ancora sostenere che la sua origine sia prevalentemente stocastica, con tutte le eventuali conseguenze (errore standard dei parametri del best-fit ed eventualmente test del χ^2). Anticipiamo qui che, purtroppo, il trattamento che abbiamo riservato all'incertezza comporta di trascurare degli effetti di origine prevalentemente sistematica che diventano evidenti, in particolari condizioni, nel campionamento di segnali variabili nel tempo, circostanza che rende un po' frustrante il lavoro compiuto.

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 [9]. Queste misure falsate contribuiscono all'istogramma, pur essendo non necessariamente rappresentative della distribuzione dei valori 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 probabilità di registrare artefatti legati alla natura periodica della d.d.p. di rete elettrica.

1. *Operazione con riferimento interno a 1.1 V*

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 $V_{max} = 1.1$ 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. 7 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 distribuzione normale. 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 scarsa stabilità meccanica, e quindi elettrica, del potenziometro.

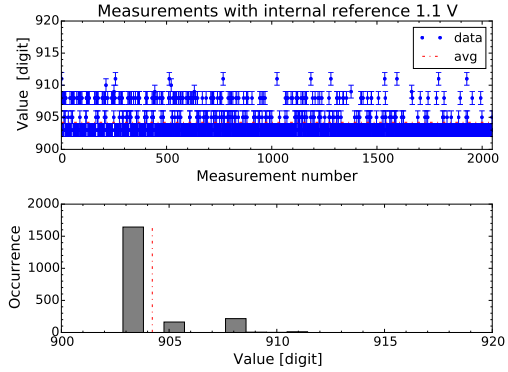

Figura 7. Analogo di Fig. 6 ottenuto utilizzando la tensione di riferimento interna $V_{ref} = 1.1$ V (nominali) e misurando con il multimetro digitale $\Delta V = (964 \pm 5)$ mV. Il valore medio digitalizzato è 904.6 digit e la deviazione standard sperimentale è 7.6 digit.

B. Campione degli intervalli di campionamento

Le acquisizioni di questa esercitazione pratica 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 con il valore nominale Δt_{nom} impostato nello script di Python (in questo esempio, $\Delta t_{nom} = 500$ μs) e di dedurre la deviazione standard sperimentale del campione degli intervalli di campionamento.

Dalle specifiche di Arduino si sa che i tempi sono determinati, e quindi anche misurati, con una risoluzione di 4 μs, decisamente peggiore di quella virtualmente consentita dalla frequenza del clock primario basato su oscillatore a quarzo da 16.000 MHz. 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. 8 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.

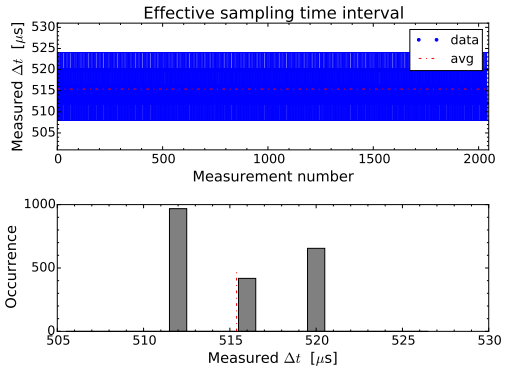

Figura 8. Esempio di analisi del campione di intervalli di campionamento Δt costruito come discusso nel testo. Il pannello superiore riporta il grafico delle misure, a cui è stata attribuita un'incertezza convenzionale ± 4 μs, il pannello inferiore riporta l'istogramma delle occorrenze. Il valore medio digitalizzato è 515.4 μs e la deviazione standard sperimentale è 3.5 μs.

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, quello che in precedenza abbiamo indicato con Δt_{dig}. Dalle misure effettuate su una specifica scheda di Arduino esso risulta di circa 15 μs. La deviazione standard sperimentale ottenuta dal campione è 3.5 μs: questo valore può essere considerato come un'indicazione della deviazione standard associata alla misura del tempo da parte di Arduino nelle condizioni del nostro esperimento. Notate tuttavia che la distribuzione mostrata nell'istogramma è anche in questo caso tutt'altro che normale. Il risultato suggerisce comunque che l'incertezza dedotta dalle specifiche (4 μs) è simile (leggermente superiore, nel nostro esempio) a quella determinata esaminando il campione.

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 e che anche voi siete invi-

tati a registrare volontariamente. 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. 6 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 e non solo quello corrispondente alla lettura nulla.

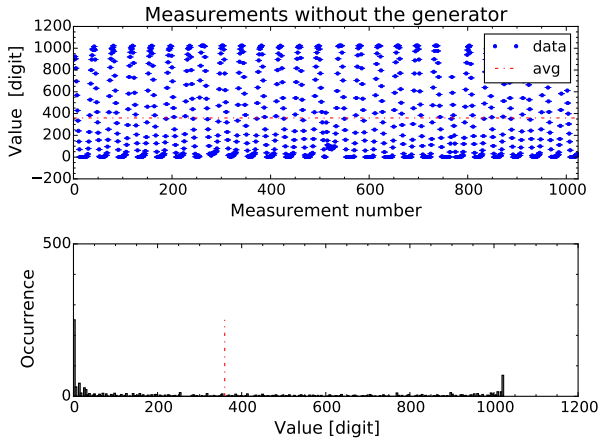

Figura 9. Analogo di Fig. 6 dove, però, il generatore è stato scollegato, lasciando "volante" la boccola collegata al pin `A0` di Arduino, ovvero *flottante* la d.d.p. in ingresso al digitalizzatore.

L'interpretazione è piuttosto immediata: quelle registrate sono letture dovute a "disturbi" che vengono raccolti 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 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* e potrebbe essere interessante, fra qualche mese, esaminare il record così costruito con il metodo della *trasformata di Fourier numerica*.

Della presenza di queste fluttuazioni 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, 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 migliore 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 è superiore rispetto a quella (già molto alta) del multimetro digitale 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 implica infatti collegare l'ingresso alla linea di massa, o di terra.

---

[1] In seguito a confusi sviluppi della situazione societaria di Arduino, esiste anche altre denominazioni per la scheda, tra cui "Genuino".

[2] Naturalmente ci sono importantissimi controesempi a questa affermazione, dovuti in particolare alla natura discreta della carica elettrica. Provate per esempio a trovare il legame tra d.d.p. e carica accumulata in un condensatore di capacità piccolissima, dell'ordine dell'aF. Tuttavia le grandezze di interesse pratico nel mondo macroscopico possono essere sicuramente considerate non discrete.

[3] È evidente che il sample-and-hold e tutto l'ambaradan necessario al suo funzionamento sono componenti cruciali di un digitalizzatore, dovendo operare su scale temporali particolarmente brevi (a questo fa riferimento l'aggettivo "istantaneo"). Probabilmente questi componenti sono i principali responsabili del comportamento "erroneo" che si riscontra quando Arduino viene impiegato per digitalizzare segnali variabili nel tempo (scalettature e andamenti irregolari).

[4] Le porte digitali di Arduino rispettano lo standard TTL (Transistor Transistor Logic). Di esso esistono in realtà diverse implementazioni sviluppatesi nel tempo, ma, in linea di massima, un segnale è considerato "alto" se la sua d.d.p. rispetto alla linea di riferimento (massa o terra) è superiore a 2.4 V, "basso" se tale d.d.p. è compresa tra 0 e 0.4 V. Quelle date nel testo sono le condizioni che si riscontrano normalmente con Arduino.

[5] Il duty-cycle rappresenta, in valore relativo o percentuale, la quantità di tempo in un singolo ciclo in cui il segnale si trova allo stato alto. Un segnale (periodico) con duty-cycle del 50%, o di 0.5, in uscita da una porta `PWM` di Arduino rappresenta di fatto un'onda quadra *simmetrica*.

[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.

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

[9] In futuro torneremo su questi "disturbi", indicandoli talvolta come *rumore* prodotto dall'interazione con l'ambiente. Molto grossolanamente, infatti, chiameremo rumore tutte le fluttuazioni dei valori letti diverse da cause fisiche che possiamo controllare.