Calibrazione di Arduino Due

francesco.fuso@unipi.it
(Dated: version 2 - FF, 27 ottobre 2023)

Questa breve nota riporta alcune generalità sull'impiego di Arduino nelle esercitazioni di Lab2, facendo particolare riferimento al modello Arduino Due. Inoltre essa discute della calibrazione del convertitore A/D di Arduino Due eseguita in diversi modi e illustra anche alcune altre osservazioni sperimentali sul comportamento del digitalizzatore. La nota può essere utile per lo svolgimento di una esercitazione "remota" dedicata alla calibrazione, anche se i dati qui considerati sono diversi rispetto a quelli distribuiti per quella esercitazione.

I. ARDUINO COME SCHEDA I/O

Nella filosofia dei nostri esperimenti, Arduino ha lo scopo principale di campionare e digitalizzare delle differenze di potenziale, operazione che in gergo si può identificare con la conversione analogico/digitale (A/D, oppure ADC, con la C che sta per Converter, o Conversion). Per questo scopo esso viene collegato a un computer che controlla l'esperimento e quindi può essere considerato come una scheda di input/output (I/O) in estensione al computer.

In termini generali, Arduino può compiere numerose operazioni sia hardware che software. Il core di Arduino Due è un microcontroller di tipo ARM a 32 bit (Atmel SAM3X8E ARM Cortex-M3) (ARM vuol dire Advanced RISC Machine, RISC sta per Reduced Instruction Set Computer) che usa la stessa architettura di base della quasi totalità dei processori per telefonini, tablet e alcuni computer. Pur non essendo assimilabile a un computer per la mancanza di alcune funzionalità basilari (banalmente, non c'è un vero e proprio sistema operativo), Arduino può essere istruito via software per compiere numerose operazioni. In particolare, il microcontroller dispone di diverse *porte* che possono ricevere segnali (d.d.p.) dall'esterno o fornire segnali (d.d.p.) all'esterno. Queste porte, accessibili tramite dei connettori a pettine disposti sul perimetro della scheda, si dividono in porte *analogiche* e *digitali*, a seconda che siano in grado di digitalizzare (o produrre) d.d.p. "analogiche", cioè che, in linea di principio, possono variare in modo continuo, o di distinguere tra i livelli "bassi" (lo 0 binario, corrispondente a una d.d.p. al di sotto di una certa soglia dipendente dallo standard) e "alti" (l'1 binario, corrispondente a una d.d.p. superiore a una soglia) di una d.d.p.

Le porte disponibili su Arduino Due sono, principalmente:

- 12 ingressi analogici che possono accettare d.d.p. comprese tra 0 e circa 3.2 V, di segno rigorosamente positivo, offrendo quindi funzionalità A/D;
- 2 uscite analogiche che forniscono una d.d.p. compresa tra circa 0.55 e 2.75 V (le funzionalità come convertitore D/A sono limitate, in particolare in termini di corrente che può essere erogata, e non faremo uso di queste porte per i nostri scopi);
- ben 54 porte digitali che possono essere configurate come input o output. I segnali digitali coinvolti hanno livello "basso" prossimo a 0 V e livello "alto" di circa 3.2 V: notate che questo standard è differente rispetto al comune standard TTL, come lo conoscerete a Lab2, in cui il livello alto è prossimo a 5 V.
- 12 di queste porte digitali supportano la modalità PWM (Pulse Width Modulation), che in qualche occasione useremo in laboratorio.

Naturalmente nei nostri esperimenti ci limiteremo a un uso minimale di Arduino, che coinvolgerà al massimo due porte analogiche e due porte digitali. Nell'esercitazione che consideriamo qui, usiamo in pratica una sola porta analogica e, in modo molto marginale, una sola porta digitale.

Coerentemente con lo spirito di semplicità e trasparenza di Lab2, e in accordo con il regime di repubblica delle banane che vige nelle nostre stanze, due porte analogiche (chiamate `A0` e `A2`) e due porte digitali (pin `5` e `7`) sono collegate a boccole volanti di diverso colore: una quinta boccola volante, di colore nero, è collegata alla massa di Arduino.

Per quanto può interessare, il microcontroller lavora a un clock di 84 MHz (il periodo risultante stabilisce, grossolanamente, il tempo minimo perché venga compiuta un'operazione elementare) e all'interno del microcontroller sono alloggiate tre memorie distinte: una, di tipo non riscrivibile (PROM - Programmable Read Only Memory), contiene l'equivalente rudimentale del sistema operativo, delle altre due tratteremo brevemente in seguito.

A. ADC in Arduino

Come già affermato, il ruolo principale che Arduino ha nei nostri semplici esperimenti è quello di campionare e digitalizzare delle d.d.p. variabili nel tempo. In termini generali, le caratteristiche di un digitalizzatore possono essere riassunte in due figure di merito principali:

- la *profondità* (o dinamica) di digitalizzazione, che indica il valore massimo del numero intero che risulta dal processo di digitalizzazione;

- il *rate di campionamento*, rappresentativo della rapidità con la quale il digitalizzatore può campionare una d.d.p. variabile.

La profondità di digitalizzazione è in genere espressa da un numero intero n, spesso definito *bit* (per ragioni storiche), tale che il numero massimo che risulta dal processo di digitalizzazione è $N = 2^n - 1$. In altre parole, e per fare un esempio che è comune ai digitalizzatori in uso nella maggior parte degli oscilloscopi digitali, un digitalizzatore da 8 bit è in grado di suddividere la d.d.p. da misurare in $2^8 = 256$ livelli distinti. Quindi il numero intero che rappresenta la grandezza analogica ha una dinamica che va da zero a $256 - 1 = 255$ unità di digitalizzazione, che in gergo chiamiamo *digit*.

Arduino Due ha una dinamica nativa di 12 bit, corrispondenti a 4096 livelli distinti: quindi il numero intero risultante dalla digitalizzazione di una d.d.p. può assumere un valore compreso tra 0 e 4095 digit.

Per quanto riguarda il rate di campionamento, argomento che non è molto rilevante per questa nota (lo sarà di sicuro in futuri esperimenti!), possiamo supporre che esso sia limitato in alto dall'intervallo di tempo finito, qui chiamato Δt_{dig}, che occorre per portare a termine una singola digitalizzazione. Questo tempo non è citato in modo esplicito nei datasheet del microcontroller, ma, come anche risulterà da questa nota e dalle vostre analisi dati, dovrebbe essere di circa $5 - 7$ μs, corrispondenti a un rate di campionamento massimo (virtuale) di oltre 100 kSa/s (si legge kilosample per secondo). Apprezzeremo il significato di questa affermazione, calandola nella realtà dei nostri esperimenti, in qualche prossima esercitazione.

II. STRATEGIA GENERALE DEGLI ESPERIMENTI CON ARDUINO, SKETCH E SCRIPT

Per funzionare come richiesto, Arduino ha bisogno di un insieme di istruzioni che, appunto, stabiliscano cosa deve essere fatto (quali porte devono essere lette o scritte, quando, etc.). Questo insieme di istruzioni ha la forma di un programma scritto in un linguaggio che è un sottoinsieme del linguaggio C, quindi non molto bello dal punto di vista estetico per la presenza di numerosi vincoli sintattici. Nella pratica, questo programma, che ha il nome gergale di *sketch*, viene scritto in formato testo su un'interfaccia (Arduino IDE - Interactive Development Environment, o semplicemente Arduino; essa può essere lanciata nei computer di laboratorio cliccando sull'icona a forma di infinito - simbolo di Arduino - nel menu applicazioni): questa interfaccia permette anche di trasferire il programma ad Arduino (l'operazione si chiama *upload* e l'icona che la permette ha la forma di una freccia).

Al termine del trasferimento, che avviene via comunicazione seriale, lo sketch si trova *residente* nel microcontroller, potendo contare su un'apposita sezione di memoria. Questa memoria è *non volatile* (NVRAM - Non Volatile Random Access Memory), ovvero di tipo *flash*: ciò significa che lo sketch rimane in Arduino finché non ne viene sovrascritto un altro. Questa memoria ha un'estensione di 512 kB, più che sufficiente per i nostri scopi tipici.

Nella pratica di laboratorio, gli sketch (come anche gli script, di cui tratteremo fra breve) si trovano nella sotto-directory `Arduini/` (notate il plurale) della directory principale dei computer: precisamente, gli sketch, che hanno estensione `.ino`, devono trovarsi in una sotto-directory che ha lo stesso nome dello sketch. Normalmente troverete gli sketch già caricati nella memoria di Arduino, che quindi sarà pronto per operare: ricordate però di collegarlo alla porta USB del computer, da cui Arduino riceve anche l'alimentazione, per evitare comportamenti erratici o indefiniti.

La comunicazione tra Arduino e il computer si basa su un protocollo molto poco efficiente. Infatti essa è di tipo seriale (emulata su una linea USB, ma pur sempre seriale), cioè i dati vengono inviati un bit alla volta. Inoltre la comunicazione seriale è normalmente poco robusta ed è in genere utile aggiungere dei tempi morti per evitare intasamenti e casini vari (a questo scopo vengono inseriti dei ritardi di ben 2 s in posizioni strategiche dei programmi). Questo rende praticamente impossibile trasferire in tempo reale (o quasi-reale) i dati digitalizzati al computer, operazione che viene invece svolta di routine negli oscilloscopi digitali che userete in futuro. La strategia utilizzata con Arduino prevede infatti che l'intero set di dati, chiamato in seguito anche *record*, venga inizialmente trasferito a una memoria (di tipo SRAM - Static Random Access Memory, cioè volatile) interna al microcontroller e quindi, al termine dell'acquisizione, tutto il record sia comunicato al computer. La memoria SRAM ha dimensioni non troppo ampie (96 kB), circostanza che pone delle limitazioni nel massimo numero di dati, o *lunghezza*, del record. Queste limitazioni sono pressoché irrilevanti per l'esperimento trattato in questa nota, ma saranno di potenziale interesse in future applicazioni.

La lentezza della comunicazione seriale è un fattore da tenere presente nell'impiego ordinario di Arduino, in particolare nel caso di record lunghi, cioè composti da qualche migliaia di dati. Tenendo conto anche dei vari ritardi strategici necessari per evitare casini, il trasferimento dati dalla memoria di Arduino al computer può richiedere diversi secondi, anche una decina.

La comunicazione con il computer si basa su uno script di Python, che ha anche il compito di avviare l'acquisizione e, quando necessario (non per l'esperimento citato in questa nota), di trasmettere ad Arduino alcuni parametri essenziali. Anche gli script si trovano già installati nei computer di laboratorio (sempre sotto-directory `Arduini/`, il nome è lo stesso degli sketch con cui devono essere usati e l'estensione è, ovviamente, `.py`): si consiglia di non modificarli o, in caso contrario, di cambiarne il nome e spostarli in altra directory. Dato che gli script adoperano diversi pacchetti di Python non necessariamente convenzionali, essi possono essere lanciati solo usando Pyzo: per aprire Pyzo è sufficiente cliccare sul-

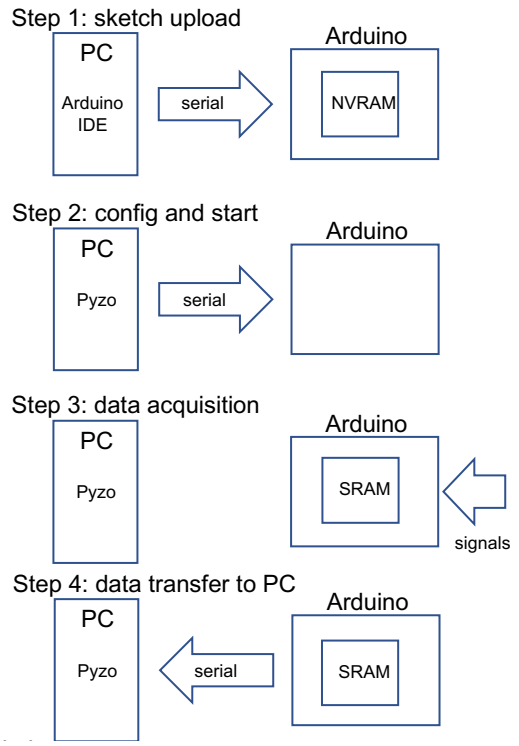

Figura 1. Diagramma semplificato della strategia di impiego di Arduino nei nostri esperimenti.

l'icona corrispondente (in genere le lettere Ze, stilizzate) nel menu delle applicazioni.

La Fig. 1 mostra un diagramma di massima delle fasi che compongono i nostri tipici esperimenti con Arduino.

A. Sketch e script `arducal`

Capire struttura di sketch e script è spesso un'operazione complicata e non particolarmente utile per gli scopi didattici del nostro corso. Tuttavia, vista la semplicità dell'esperimento descritto in questa nota, vale sicuramente la pena di soffermarsi su questi dettagli.

Esaminiamo per primo lo sketch, cioè il programma che permette ad Arduino di compiere le operazioni desiderate. Il linguaggio usato è simil-C, per cui ogni istruzione (a parte alcune eccezioni) deve terminare con un `;`.

La struttura dello sketch, riportato qui nel seguito, prevede un blocco iniziale in cui vengono definite, ed eventualmente inizializzate, variabili e array usati da Arduino: tutto mi pare abbastanza comprensibile, a parte la distinzione tra varie tipologie (`int`, `long`, etc.), che ha però prevalentemente lo scopo di garantire un po' di back-compatibility con Arduino Uno. Infatti in Arduino Due praticamente tutte le variabili e gli array di nostro interesse sono rappresentati da numeri interi a 4 byte.

Dopo il blocco definizioni ha inizio la parte di inizializzazione, che comincia con l'istruzione `void setup()` ed è compresa tra parentesi graffe. Qui, con istruzioni che sono abbastanza comprensibili, si inizializza la porta seriale (a 119200 baud, unità di misura ereditata dal mondo delle telescriventi! Nel nostro caso dovrebbe essere 1 baud = 1 bit/s), se ne pulisce il buffer, si pone a livello alto ("si accende") la porta digitale `digitalPin_uno` (che, nel blocco definizioni, è identificata con la porta `7`) per gli scopi che discuteremo nel seguito, e si istruisce il digitalizzatore a impiegare una dinamica di 12 bit, che è quella nativa.

Quindi compare il blocco del programma vero e proprio (inizia con l'istruzione `void loop()` ed è contenuto tra parentesi graffe). Il programma ha inizio nell'istante in cui Arduino riceve dei caratteri via porta seriale: se questo si verifica, la variabile intera `start`, che funge da "flag", viene posta pari a `1`. Ciò consente di eseguire le istruzioni scritte nel seguito (per `start = 0` si esce dal ciclo void e il programma termina): dopo un ritardo strategico di 2 s, si chiede ad Arduino di fare un paio di misure che poi vengono cancellate. Il motivo è quello di minimizzare la comparsa di artefatti (dati non corrispondenti alla realtà, cioè non dovuti a segnali in ingresso alla porta analogica `A0`, quella usata in questo esperimento) creati, per esempio, da qualche impulso di corrente che dovesse circolare in Arduino nella fase di lettura della porta seriale. Infatti, come avrete ampio modo di scoprire durante questo anno accademico, gli impulsi di corrente possono "accoppiarsi" con gli ingressi analogici (non solo quelli di Arduino!) dando luogo a *spikes* di d.d.p.

Quindi la variabile intera `StartTime`, che gioca il ruolo dello zero nella misura dei tempi, viene azzerata e comincia un ciclo `for` per il campionamento e la digitalizzazione di 8192 dati. Guardate bene come avviene la costruzione dei dati: nell'array `t[i]` viene consecutivamente registrato il tempo, in microsecondi, che Arduino conta. A questo scopo viene letto il contenuto del timer interno di Arduino (l'istruzione corrispondente è `micros()` e, ovviamente, il numero generato è un intero in unità di microsecondi) e viene sottratta la variabile `StartTime` per tenere conto dello zero dei tempi. Quindi viene interrogata la porta `A0` e il suo valore digitalizzato viene registrato nell'array `V[i]`. Infine, viene inserita un'istruzione che fa attendere un ritardo, espresso in microsecondi e contenuto nella variabile `delays` (inizializzata a 100 nel blocco definizioni, per questo esempio). Apprezzerete meglio nel futuro l'utilità di questo ritardo, che in gergo chiameremo Δt_{nom}: esso, in sostanza, consente di compiere un'acquisizione della durata complessiva necessaria per seguire un qualche fenomeno di interesse. Poiché la lunghezza del record è (in ogni caso, non solo con Arduino!) limitata, occorre aspettare per garantirsi che i dati acquisiti riescano a coprire l'intervallo temporale di interesse.

Nel caso trattato in questa nota, dove si misurano d.d.p. continue (virtualmente costanti nel tempo), la scelta di Δt_{nom} non è critica. Tenete presente che, per $\Delta t_{nom} = 100$ μs, le 8192 misure del ciclo richiedono un

tempo dell'ordine di $8192 \times 100\ \mu s \simeq 1$ s (arrotondando grossolanamente).

Terminato il ciclo di misura, i dati sono pronti per essere trasferiti al computer tramite porta seriale: un ulteriore ciclo, piuttosto auto-esplicativo, provvede allo scopo. Alla fine la variabile `start` viene azzerata e il ciclo void si interrompe.

```
// Blocco definizioni
const unsigned int analogPin=0; // Definisce la porta A0 per la lettura
const int digitalPin_uno=7; // Definisce la porta 7 usata come output
int i; // Definisce la variabile intera i (contatore dei cicli di lettura e scrittura)
int delays = 100; // Definisce la variabile intera delays (ritardo in microsecondi fra digitalizzazioni)
int V[8192]; // Definisce l'array intero V che contiene le letture
long t[8192]; // Definisce l'array t che contiene il tempo della lettura, in microsecondi
unsigned long StartTime; // Definisce la variabile StartTime
int start=0; // Definisce la variabile start (usata come flag)
int dummy; // Definisce una variabile dummy usata per scaricare la porta seriale

// Istruzioni di inizializzazione
void setup()
  {
   Serial.begin(119200); // Inizializza la porta seriale a 119200 baud
   Serial.flush(); // Pulisce il buffer della porta seriale
   digitalWrite(digitalPin_uno,HIGH); // Pone digitalPin_uno a livello alto
   analogReadResolution(12);// Istruisce il digitalizzatore di usare la risoluzione, o dinamica, a 12 bit
  }

// Istruzioni del programma
void loop()
  {
    if (Serial.available() >0) // Controlla se il buffer seriale ha qualcosa
      {
      dummy=Serial.read(); // Legge la parte seriale e la svuota se c'e' qualcosa
      start=1; // Nel caso ci sia qualcosa, 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
    delay(2000); // Aspetta 2000 ms per evitare casini

     for(i=0;i<2;i++) // Fa un ciclo di due letture a vuoto per evitare artefatti
       {
        V[i]=analogRead(analogPin);
       }
    StartTime=micros(); // Misura il tempo iniziale con l'orologio interno
    for(i=0;i<8192;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<8192;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, cioe' fa terminare il programma
    Serial.flush(); // Pulisce il buffer della porta seriale (si sa mai)
}
```

Lo script di Python progettato per lavorare assieme allo sketch, anch'esso riportato nel seguito, è estremamente semplice e dovrebbe risultarvi facilmente comprensibile. In sostanza, all'inizio crea un file (in formato `.txt`) destinato ad accogliere i dati trasferiti da Arduino, che avrà l'aspetto di due colonne (tempo in μs e valore digitalizzato in digit), e lo predispone per la scrittura. Quindi lo script provvede a inviare un carattere a caso (il carattere è `G`) via porta seriale ad Arduino e si pone in attesa per la ricezione dei dati: non appena questi arrivano, vengono scritti sul file attraverso un ciclo `for`. Inoltre, per velocizzare le operazioni, i dati trasferiti da Arduino vengono anche decodificati opportunamente e messi in un array (`runningddp`), che viene analizzato per determinarne valore medio e standard deviation (entrambi scritte sulla console). Unici elementi un minimo esotici dello script sono le librerie `serial` e `time`, che servono per gestire la comunicazione con la porta seriale e per creare i ritardi strategici necessari per evitare casini.

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

Directory='home/studenti/dati_arduino/'   # nome directory dati  << DA CAMBIARE SECONDO NECESSITA'

FileName=Directory+'datacal.txt'  # parte comune nome file << DA CAMBIARE SECONDO NECESSITA'

outputFile = open(FileName, "w" ) # apre file dati predisposto per scrittura

print('Please wait') # scrive di aspettare sulla console

ard=serial.Serial('/dev/ttyACM0',119200)  # apre porta seriale (la sintassi dipende dal sistema operativo!)

time.sleep(2) # ritardo strategico di 2 s
print('Start Acquisition') # scrive sulla console (terminale)

ard.write(b'G') # scrive un carattere sulla porta seriale; l'istruzione b indica che e' un byte
time.sleep(2) # ritardo strategico di 2 s

 # loop lettura dati da seriale (8192 coppie di dati: tempo in us, valore digitalizzato di d.d.p.)
runningddp=numpy.zeros(8192) # prepara il vettore per la determinazione della ddp media e std

for i in range (0,8192):
        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 nell'array

ard.close() # chiude la comunicazione seriale con Arduino

avgddp=numpy.average(runningddp) # analizza il file 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, ciao') # scrive sulla console che ha finito e vi saluta
```

III. "CALIBRAZIONE" DI ARDUINO

Affinché la lettura digitalizzata possa essere espressa in unità fisiche occorre una conversione. Essa può essere realizzata applicando un qualche modello, cioè una qual-

che semplice espressione analitica, alla lettura digitalizzata: i fattori che entrano in questa espressione analitica devono generalmente essere calibrati, cioè determinati attraverso specifiche misure. Poiché le misure sono affette da una data incertezza, la calibrazione è sempre associata a incertezza, che dà luogo all'onnipresente *incertezza di calibrazione*.

Alcune premesse rilevanti:

- La calibrazione qui presentata come esempio si riferisce a *una* specifica scheda di Arduino impiegata in specifiche condizioni: le inevitabili differenze di costruzione fanno sì che la calibrazione debba essere ripetuta per ogni singola scheda. Anche le condizioni di operazione, in particolare la temperatura, influiscono sulla calibrazione.

- Idealmente una procedura di calibrazione dovrebbe essere basata su una sorgente calibrata di d.d.p. (uno *standard*); nel caso in cui una tale sorgente sia indisponibile, come nel nostro laboratorio, la calibrazione può essere effettuata *per confronto*, cioè confrontando i risultati della misura fatta da Arduino con quelli di misure eseguite con altri strumenti di cui è nota l'accuratezza. Questa è la procedura qui seguita.

- Quanto presentato si basa strettamente sull'uso della strumentazione standard del laboratorio didattico (multimetro, generatori). Sarebbe naturalmente possibile impiegare strumentazione più sofisticata, che potrebbe consentire un'accresciuta accuratezza di calibrazione.

- Aspetto molto rilevante: poichè le sorgenti impiegate non sono generatori calibrati, le incertezze determinate nella procedura di calibrazione risentono anche delle fluttuazioni nel funzionamento dei generatori e della presenza di eventuali segnali spuri raccolti dall'ambiente. Dunque esse non dovrebbero essere attribuite unicamente al digitalizzatore, come in realtà implicitamente faremo: così si compie una sorta di sovrastima dell'incertezza di calibrazione, che in questa sede non siamo in grado di valutare quantitativamente.

- Fortunatamente, nella maggior parte delle situazioni in cui useremo Arduino potremo trascurare la calibrazione e le relative incertezze. Saremo infatti interessati a ricostruire andamenti (tipicamente in funzione del tempo) dei segnali, per i quali la conversione da digit a unità fisiche non è rilevante. In tali situazioni sarà eventualmente di interesse conoscere l'incertezza statistica (grossolanamente, di digitalizzazione), argomento a cui può contribuire quanto discusso nel seguito di questa nota.

In generale possiamo ipotizzare due metodi per la calibrazione, uno più sbrigativo e necessariamente inaccurato, l'altro più corretto dal punto di vista scientifico.

A. Calibrazione "alternativa"

Il metodo di calibrazione che chiamiamo, in gergo, alternativa si basa sulla circostanza (ragionevole, ma non documentata in modo affidabile) che la massima d.d.p. digitalizzabile, quella corrispondente al livello 4095 digit, sia pari alla d.d.p. erogata in uscita su una qualsiasi porta digitale che si trova a livello alto. Inoltre il metodo si fonda sull'ipotesi di linearità (senza offset!) del digitalizzatore, cioè si suppone il modello

$$V_V = \xi V_{dig} \ , \tag{1}$$

con V_V e V_{dig} d.d.p. in unità fisiche e in digit, rispettivamente, e ξ fattore di conversione.

Nell'esempio considerato si misura su una delle porte digitali di Arduino (la porta 7, opportunamente posta a livello alto dallo sketch) $V_{max} = (3.24 \pm 0.03)$ V, da cui, nelle ipotesi enunciate,

$$\xi = \frac{V_{max}}{4095} = (791 \pm 6) \ \mu\text{V/digit} \ . \tag{2}$$

Osservate che l'intervallo di tensione corrispondente a un singolo digit, inferiore al mV (un valore piccolino per gli standard di un laboratorio di circuiti elettrici e, soprattutto, compatibile con l'entità di tante fluttuazioni e rumori), testimonia la buona dinamica di digitalizzazione di Arduino Due.

Poiché la calibrazione si basa su una singola misura, l'incertezza di calibrazione è, in termini relativi, pari all'incertezza di misura di V_{max}, cioè $\delta\xi/\xi = \delta V_{max}/V_{max}$, dove il simbolo δ indica un'incertezza. Nelle nostre condizioni ordinarie, tale incertezza relativa è generalmente dominata dagli effetti non statistici sulla misura delle d.d.p. con il multimetro.

IV. CALIBRAZIONE PER PUNTI

Il metodo di calibrazione per confronto più immediato consiste nell'inviare a una porta analogica di Arduino una d.d.p. che può essere variata in un certo range, compatibile con le richieste di Arduino Due (dunque compresa tra 0 e circa 3.2 V), acquisire questa d.d.p. con Arduino e, simultaneamente, con un multimetro. Notate che, vista la grande resistenza di ingresso del multimetro (digitale) e delle porte di Arduino, la simultaneità si realizza collegando in parallelo il multimetro alla porta di Arduino. I dati acquisiti per un certo numero di regolazioni della d.d.p. possono quindi essere fittati secondo un'opportuna funzione modello allo scopo di ricavare i fattori di conversione.

Poiché Arduino produce un record di dati, è possibile costruire un campione, i cui valore medio e deviazione standard sperimentale forniscono la lettura digitalizzata e la sua incertezza. Oltre alle inevitabili fluttuazioni della d.d.p. in ingresso, la misura digitalizzata può essere affetta anche dall'accoppiamento con segnali spuri, in

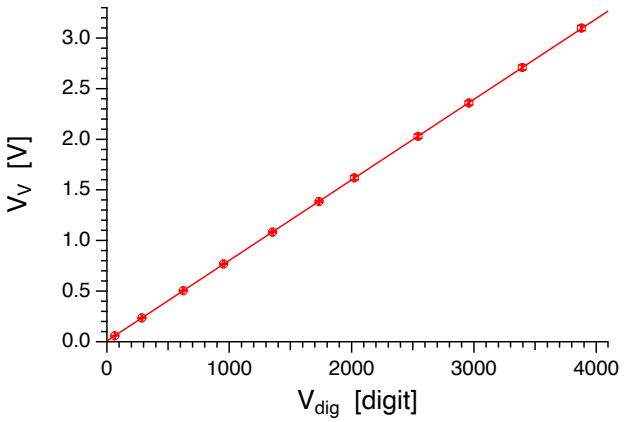

Figura 2. Valori della d.d.p. V_V applicata ad Arduino in funzione della lettura digitalizzata V_{dig}: la linea continua rappresenta il best-fit discusso nel testo.

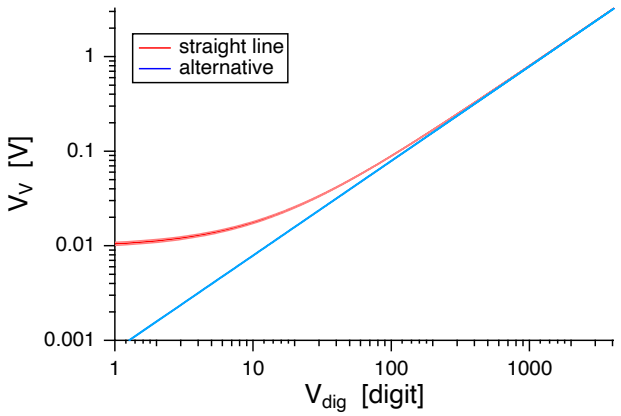

Figura 3. Confronto tra le previsioni basate sulla calibrazione alternativa, in blu, e quelle secondo l'Eq. 3. I grafici sono costruiti tenendo conto delle incertezze (e della covarianza nel caso del modello a retta), da cui lo spessore delle linee. Notate la rappresentazione bilogaritmica, usata per evidenziare le differenze per bassi valori della d.d.p. misurata.

particolare nella forma di spikes. Il processo di media può rendere la misura sufficientemente robusta nei confronti di questi effetti spuri, che costituiscono una forma di *rumore*.

A titolo di esempio, vengono qui riportate alcune misure che sono *diverse* (anche registrate e ottenute in maniera diversa, per esempio qui ho collegato Arduino a un computer portatile, alimentato a batteria e non connesso alla terra!) rispetto ai dati disponibili per un'esercitazione in remoto che vi verrà proposta a breve. Per queste misure la d.d.p. è stata creata usando un generatore continuo e un partitore in cui era presente una resistenza variabile (un *potenziometro*) montata nella configurazione di partitore di tensione.

La Fig. 2 riporta le misure e un best-fit secondo una retta con un offset, tale cioè che

$$V_V = a + bV_{dig} \ , \tag{3}$$

con a e b fattori di conversione. I risultati del best-fit, condotto usando l'errore efficace e la scelta `absolute_sigma = False` per la presenza di errori non statistici nella misura con il multimetro, sono

$$\begin{aligned} a &= (9.7 \pm 0.4) \text{ mV} \\ b &= (795.4 \pm 0.4) \ \mu\text{V/digit} \\ \text{norm. cov.} &= -0.65 \\ \kappa^2/\text{ndof} &= 0.15/9 \ ; \end{aligned}$$

il fattore di conversione a, che rappresenta l'offset, non è compatibile con zero, in accordo con le specifiche del microcontroller di Arduino (come ci è già capitato di osservare, in elettronica è piuttosto difficile ottenere uno zero). Tuttavia, esso è così piccolo che il coefficiente angolare della retta, parametro b, risulta compatibile con il fattore di conversione ξ determinato in modo alternativo.

I dati di Fig. 2 sono stati anche usati in un best-fit con una funzione quadratica del tipo

$$V_V = a' + b'V_{dig} + cV_{dig}^2 \tag{4}$$

ottenendo i risultati

$$\begin{aligned} a' &= (10.0 \pm 0.4) \text{ mV} \\ b' &= (795 \pm 1) \ \mu\text{V/digit} \\ c &= (0.3 \pm 0.4) \text{ nV/digit}^2 \\ \kappa^2/\text{ndof} &= 0.14/8 \ . \end{aligned}$$

Si vede come il termine di secondo grado, parametro c del fit, sia compatibile con lo zero e che, inoltre, il $\kappa^2_{rid} = \kappa^2/\text{ndof}$ non cali, a dimostrazione che il modello lineare è più appropriato, almeno per la specifica scheda Arduino Due esaminata.

La presenza di $a \neq 0$ rende ovviamente incompatibile la calibrazione secondo la funzione modello in Eq. 3 con quella alternativa, la cui funzione modello è in Eq. 1. Le previsioni basate sui due modelli sono confrontate in Fig. 3, dove si è tenuto conto delle incertezze (e della covarianza, nel caso del modello a retta con offset) per ottenere delle "bande di confidenza", all'interno delle quali cade la previsione. Per $V_V \lesssim 0.1$ V, cioè $V_{dig} \lesssim 200$ digit, la calibrazione alternativa fornisce valori sistematicamente sottostimati. Però è interessante notare come, per d.d.p. sufficientemente alte, le previsioni siano pressoché indistinguibili tra di loro, come atteso dato che ξ è compatibile con b. Come ovvio dato l'impiego di un best-fit, e quindi di un set di misure, l'incertezza relativa con cui è determinato b è oltre un ordine di grandezza migliore di quella con cui è noto ξ, che era stato determinato da una singola misura.

V. DISTRIBUZIONI

Le acquisizioni con Arduino sono sostanzialmente dei campionamenti, cioè il segnale viene digitalizzato in tanti

intervalli di tempo successivi. Se si invia una d.d.p. costante, o supposta costante (cioè si trascurano le sue inevitabili fluttuazioni e l'accoppiamento con segnali spuri), l'analisi statistica del campione, o record, così acquisito può dare informazioni sull'incertezza stocastica della digitalizzazione.

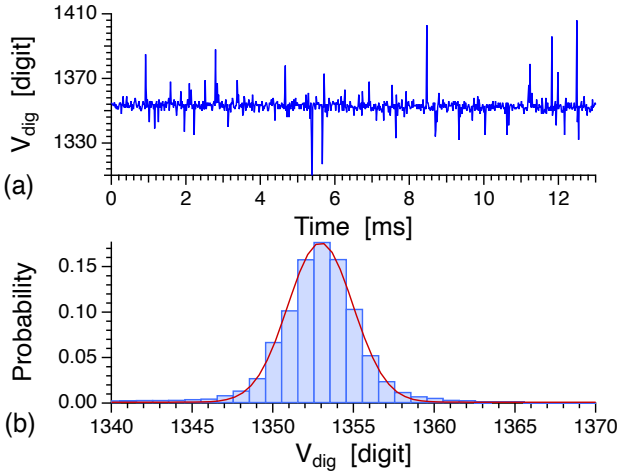

Figura 4. Acquisizione con Arduino di una d.d.p. costante $\Delta V = (1.084 \pm 0.006)$ V (a) e istogramma delle probabilità di occorrenza normalizzate (b), con sovrapposta la linea continua del best-fit, in rosso, secondo una funzione Gaussiana. L'acquisizione è stata eseguita con un tempo di campionamento nominale di 20 μs.

La Fig. 4(a) riporta un record da 8192 punti acquisito inviando ad Arduino una d.d.p. $\Delta V = (1.084 \pm 0.006)$ V (l'intervallo di campionamento nominale è $\Delta t_{nom} = 20$ μs): si vede come la misura, benché affetta dalla presenza di sporadici spikes, si mantenga attorno al valore $< V_{dig} > \simeq 1353$ digit, in accordo con il risultato di Fig. 2. La distribuzione delle probabilità di occorrenza dei valori digitalizzati, normalizzata al numero di punti del campione, è mostrata sotto forma di istogramma con bin di ampiezza unitaria in Fig. 4(b), che contiene oltre il 97% delle misure (gli spikes cadono al di fuori del range considerato). Osservate che la frequenza con cui gli spikes appaiono nei record può essere funzione delle condizioni di operazione: tali spikes possono infatti essere raccolti come rumore dall'ambiente esterno e la raccolta dipende da tantissimi fattori che non possiamo modellare (quale banco usate, quale generatore impiegate e come esso è collegato ad Arduino, quale computer state usando e se esso è collegato alla terra dell'impianto elettrico, se c'è qualche radio libera che sta trasmettendo, o magari qualche telefonino, etc. - tutti aspetti che avremo modo di considerare meglio durante l'anno). Sovrapposto all'istogramma è un best-fit secondo una funzione Gaussiana con larghezza rms $\sigma = (2.02 \pm 0.02)$ digit. Questo valore, che tende leggermente ad aumentare con ΔV, e quindi $< V_{dig} >$, può essere considerato rappresentativo della deviazione standard sperimentale del campione di misure fatto con Arduino Due. Naturalmente questa affermazione dovrebbe essere presa con le molle, poiché il risultato dipende da tanti fattori esterni e, come già affermato, tiene conto anche delle fluttuazioni del segnale prodotto (dovute al generatore e all'interazione con l'ambiente).

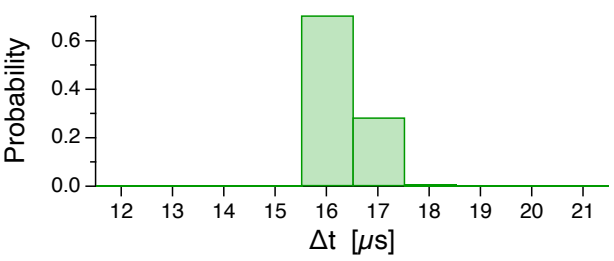

Figura 5. Istogramma delle probabilità di occorrenza normalizzate per l'intervallo temporale di campionamento, estratto da un record di 8192 punti acquisito con un intervallo nominale $\Delta t_{nom} = 10$ μs.

Una simile analisi statistica può in linea di principio essere condotta anche sugli intervalli di campionamento, cioè gli intervalli temporali tra un campionamento e il successivo. Nella nostra strategia di implementazione, l'intervallo di tempo "effettivo" tra un campionamento e il successivo è dato da $\Delta t = \Delta t_{dig} + \Delta t_{nom}$, cioè dalla somma del tempo che occorre per portare a termine una singola digitalizzazione e il ritardo introdotto via software nel ciclo di acquisizione. In questo contesto è possibile che ci siano degli ulteriori ritardi aggiuntivi, dovuti per esempio a qualche latenza del microcontroller (nella sua vita, può essere che il microcontroller si dedichi temporaneamente ad altro che non sia eseguire le istruzioni dello sketch).

Estraendo da un record di 8192 dati (si intende dati di tempi, corrispondenti all'array `t[i]` prodotto da Arduino) l'array delle differenze di tempo di campionamento Δt, e graficando questo array di differenze sotto forma di istogramma, si ottiene quanto mostrato ad esempio in Fig. 5, corrispondente a un valore nominale $\Delta t_{nom} = 10$ μs. Come si può immediatamente notare, solo due bin dell'istogramma sono popolati per cui la risoluzione temporale con cui vengono letti i tempi, 1 μs, è insufficiente per ricostruire la distribuzione. Osservate che è comunque possibile associare una media e una deviazione standard sperimentale al record, calcolata su tutti gli intervalli temporali del record stesso: nell'esempio considerato si ottiene $\Delta t = (16.3 \pm 0.5)$ μs. Tenendo conto del valore nominale impostato via software, $\Delta t_{nom} = 10$ μs, il tempo di digitalizzazione risulta $\Delta t_{dig} = (6.3 \pm 0.5)$ μs.

Notate che:

- Non essendo stato possibile ricostruire sperimentalmente la distribuzione, l'incertezza ± 0.5 μs deve essere intesa come una stima, presumibilmente una sovrastima, dell'effettiva deviazione standard nella misura dei tempi;

- In ogni caso, essa non può fornire alcuna informazione sull'accuratezza temporale del campionamen-

to. Per questo sarebbe infatti necessario confrontare la misura con quella di un altro "orologio" di calibrazione nota, obiettivo che è al di fuori degli scopi di questa nota (eh, sarebbe molto carino farlo perché potreste usare uno strumento concettuale parecchio evoluto, si chiama *Allan deviation*, e scoprire che non è troppo difficile avere orologi molto accurati, per esempio un ricevitore GPS può fornire una *base dei tempi* mooolto accurata: infatti nei satelliti GPS sono presenti degli orologi atomici, il cui segnale viene trasmesso in qualche modo ai ricevitori). In ogni caso, poichè la base dei tempi di Arduino utilizza un auto-oscillatore al *quarzo* e poiché per questi sistemi l'accuratezza tipica è dell'ordine di alcune decine di ppm (parti per milione), o migliore, si può ipotizzare che, almeno per la misura di intervalli temporali sufficientemente brevi, l'errore di calibrazione della base dei tempi produca effetti trascurabili rispetto all'incertezza statistica.

- In termini generali e un po' più sottili, il tempo contenuto nell'array `t[i]` di Arduino è stabilito dal software, cioè non è effettivamente misurato. In altre parole, il software stabilisce un ritardo con una risoluzione di 1 μs e legge gli istanti di tempo con la stessa risoluzione di 1 μs, argomento che rende poco applicabile il concetto di deviazione standard sperimentale.

VI. RIASSUNTINO: CHE INCERTEZZE USARE CON ARDUINO (DUE)

Riassumiamo in termini pratici quanto possiamo imparare dalle misure presentate in questa nota (e dalle conclusioni che trarrete voi analizzando i dati di una prossima esercitazione in remoto).

Premessa:

- nella maggior parte dei casi di impiego pratico di Arduino potremo disinteressarci della calibrazione del digitalizzatore e quindi dell'incertezza non statistica ad essa associata. Infatti saremo spesso interessati a seguire degli andamenti in funzione del tempo, dove non sarà affatto necessario, anzi sarà proprio inappropriato, convertire le letture digitalizzate della d.d.p. in unità fisiche.

Le incertezze da considerare saranno quindi di origine prevalentemente statistica, per cui `absolute_sigma` sarà `True` nei best-fit: in queste condizioni dovremmo aspettarci un $\chi^2_{rid} \simeq 1$ e vedremo se questo si verificherà (avremo valide interpretazioni se non si verificherà!). Spesso, inoltre, potremo ripetere diverse volte, in maniera automatica e periodica, la stessa misura, potendo quindi ottenere un valore medio sperimentale, per cui:

- spesso potremo stimare l'incertezza statistica dal campione di misure come deviazione standard sperimentale o, meglio, deviazione standard della media.

Nei casi in cui non potremo fare uso di medie, e in generale per quello che riguarda le misure di tempo, potremo sfruttare quanto presentato in questa nota (e quanto concluderete voi con le vostre analisi) e quindi porre, in modo convenzionale:

- una deviazione standard di *qualche* (un paio o qualcuno di più, vedremo in funzione delle diverse condizioni di operazione) digit per la misura digitalizzata delle d.d.p.;

- un'incertezza statistica di ± 1 μs per la misura dei tempi (che, nella pratica, si ridurrà a trascurare l'incertezza sui tempi nelle nostre analisi - quindi niente errore efficace nei best-fit!).

Richiamiamo ancora il carattere *convenzionale* di queste affermazioni, riservandoci di modificarle sulla base degli esperimenti che svolgeremo, e sottolineiamo come stiamo così trascurando l'accuratezza (errore non statistico) sulla misura dei tempi.