UNIVERSITÀ DI PADOVA **FACOLTÀ DI INGEGNERIA**

TESI DI LAUREA

INTEGRAZIONE DEL MOTION TRACKER TRIVISIO COLIBRÌ IN AMBIENTE PURE DATA

Laureando: *Alessio Fusaro*

Relatore: Prof. Federico Avanzini
Correlatore: Ing. Michele Geronazzo

Corso di Laurea Triennale in Ingegneria dell'Informazione

27 Novembre 2012

Anno Accademico 2012/2013

Prefazione

Il concetto di AUDIO 3D, che assume un ruolo centrale all'interno del progetto in cui si inserisce il lavoro svolto, trova le sue origini negli studi eseguiti all'inizio del Novecento ad opera del fisico inglese John Strutt, meglio conosciuto come Lord Rayleigh. La sua Duplex Theory esposta nel libro "On our perception of sound direction" è ancora considerata quella fondamentale sulla localizzazione del suono. Lo sviluppo delle tecniche binaurali per la riproduzione della spazialità del suono, unito all'utilizzo di dispositivi elettronici d'avanguardia a prezzi accessibili, permette ora un elevato livello di interattività tra la scena virtuale sonora ricreata e l'utente.

Il lavoro di tesi svolto riguarda l'utilizzo del sensore di movimento Triviso Colibrì come dispositivo di head-tracking per il tracciamento della posizione della testa dell'ascoltatore all'interno di progetti relativi alla spazializzazione del suono in ambiente di programmazione Pure Data.

Nel primo capitolo verranno descritti i concetti di spazializzazione e localizzazione del suono ed approfonditi nel dettaglio scopi e caratteristiche dell'head-tracking.

Nel secondo capitolo viene descritto il sensore di movimento Trivisio Colibrì su cui è stato svolto il lavoro di tesi, in particolare dopo una descrizione delle principali caratteristiche, ne verranno descritti il funzionamento e le modalità di gestione delle risorse dal computer mediante le API in dotazione.

Nel terzo capitolo verranno inizialmente introdotti l'ambiente Pure Data ed il metodo di programmazione utilizzato. In seguito verrà analizzato il lavoro svolto consistente in una external in grado di stabilire la connessione tra computer e sensore e di gestirne i dati rilevati necessari agli scopi previsti.

Il quarto capitolo conclude la tesi presentando le considerazioni finali sull'operato e delineando i possibili sviluppi futuri dovuti all'utilizzo di sensori di movimento. In appendice viene messo a disposizione il codice C++ che descrive il funzionamento dell'external realizzata.

Sommario

Lo sviluppo di modelli matematici e di elaborazione del segnale in grado di descrivere la soggettiva percezione del suono da parte dell'utente (basato su tecniche binaurali ed HRTF), unito alla disponibilità di strumenti elettronici d'avanguardia a prezzi accessibili, permette la realizzazione di realtà virtuali sempre più complete. L'avanzamento della ricerca in questa direzione ha portato alla nascita dei cosiddetti VAS, Virtual Auditory Scene, mediante i quali l'ascoltatore può immergersi in vere e proprie scene virtuali sonore interattive in cui viene realizzata la spazializzazione del suono attraverso l'applicazione ad esempio della sintesi binaurale.

La grande potenzialità di calcolo raggiunta dagli elaboratori in commercio ed i progressi della ricerca permettono ora più di prima la considerazione in questo campo di fattori sempre più numerosi ed eterogenei come la natura del suono, la forma della stanza simulata, la fisionomia dell'ascoltatore e la sua posizione per il raggiungimento nuovi livelli di immersione sonora.

Il lavoro di tesi svolto è rivolto in particolare all'incremento del grado di interattività di questi sistemi: l'head-tracking, realizzato mediante il sensore di movimento Trivisio Colibrì Wireless, introduce un fattore di dinamicità all'esperienza di ascolto che aiuta a migliorare la localizzazione dei suoni. Viene proposta in questa tesi un'integrazione di tale sensore in Pure Data, ambiente di programmazione per l'elaborazione di segnali audio.

Ringraziamenti

Al Prof. Avanzini,
per avermi dato la possibilità di svolgere
questo lavoro di tesi e collaborare con il
Sound & Music Computing Group dell'Università di Padova,

all'Ing. Geronazzo,
per la grande cortesia e disponibilità con le quali
mi ha seguito durante gli ultimi mesi.

A Nicolò e Roberta,
per la pazienza avuta nei miei confronti durante questi tre anni
nonostante i miei tempi e i miei metodi di studio discutibili,

a Nicolas,
per aver sostenuto con entusiasmo il mio percorso universitario dal primo giorno
e per i ricorrenti aiuti economici,

a GianCarlo e AnnaMaria,
per il costante sostegno dimostratomi in questa parte della mia vita.

A Giulia, Luca e Stefano,
per il tempo passato assieme tra i banchi
ed i fondamentali corsi di recupero on-demand,

a Fabio e Andrea,
per non avermi abbandonato nonostante tutte le volte
in cui mi sono chiuso in casa senza farmi sentire.

Dedico infine con affetto questo lavoro ed i miei anni di studio alla memoria di Nonna Argìa,
per sempre nel mio cuore, immortale modello di vita.

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Introduzione

1.1 Audio 3D

Il concetto di *spazializzazione* sonora descrive la capacità dell'apparato uditivo umano di attribuire ai suoni percepiti la determinata posizione nello spazio tridimensionale dalla quale provengono, consentendone quindi la cosiddetta *localizzazione*. Nell'approccio descritto si assume che l'intera informazione elaborata dall'apparato uditivo sia contenuta nella pressione acustica del segnale all'ingresso del timpano. Con queste assunzioni due eventi sonori che producono la stessa pressione acustica ai timpani vengono localizzati nello stesso identico modo.

1.1.1 Percezione del suono

A seconda della distanza della sorgente dall'orecchio si distinguono due situazioni:

- *Far Field*, quando la sorgente è posta a più di un metro dall'orecchio il fronte delle onde acustiche può essere approssimato con un piano (come se la sorgente fosse posta a distanza infinita).

- *Near Field*, quando la sorgente è posta a meno di un metro dall'orecchio il fronte d'onda non può più essere supposto piano e si rende necessaria l'introduzione di modelli matematici più raffinati e completi.

La localizzazione è resa possibile dalla presenza di due punti di ascolto distinti nel ricevitore costituiti dalle due orecchie separate dalla testa. La stessa onda sonora proveniente da una sorgente nello spazio giunge ai due timpani in tempi differenti e dopo aver subito attenuazioni diverse. Considerando il modello semplificato mostrato in Figura 1.1, nel quale la testa è rappresentata da una sfera e le onde sonore hanno fronte d'onda piano, si definiscono quindi:

- *Interaural Time Difference* (ITD), che rappresenta il ritardo temporale tra i due istanti di ricezione dello stesso segnale, dovuto alla velocità finita del suono ed alla distanza non nulla tra i due timpani

- *Interaural Level Difference* (ILD), che rappresenta la differenza di intensità del suono ricevuto ai due timpani dovuto alla presenza dell'ostacolo solido tra essi interposto costituito dalla testa.

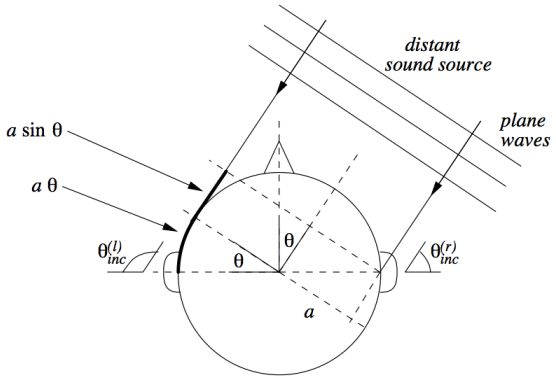

Figura 1.1: *Modello semplificato della testa*

Mentre l'ITD può essere considerato come parametro indipendente dalla frequenza del segnale, l'ILD al contrario deve tener conto della differenza di attenuazione dovuta alla testa di un'onda ad alta frequenza piuttosto che di una a bassa frequenza. Infatti se la lunghezza d'onda è molto maggiore del raggio della testa quest'ultima non è in grado di attenuarne sensibilmente l'intensità.

Le onde sonore generate da una sorgente inoltre giungono ai due timpani più volte dopo aver subito fenomeni di *riflessione* e di *diffrazione* dovuti alla presenza di ostacoli come il corpo dell'individuo oltre a quelli di natura ambientale.

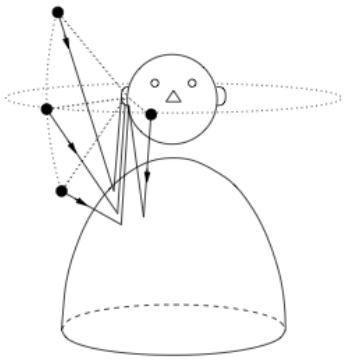

Figura 1.2: *Contributo del busto e delle spalle*

Il *busto* e le *spalle* in particolare contribuiscono introducendo nuove riflessioni ed attenuando in maniera considerevole le onde sonore provenienti dal basso anche a bassa frequenza, ecco

perché gli effetti della loro presenza si sentono in particolar modo al variare dell'altezza della sorgente quando questa è inferiore a quella delle spalle. Un riscontro grafico di quanto detto finora può essere osservato nella Figura 1.2.

L'ultimo fattore che contribuisce alla localizzazione dei suoni è introdotto dalla forma dell'*orecchio esterno*. Questa infatti da origine a fenomeni di riflessione ed interferenza che avvengono dentro all'orecchio prima del timpano. Il segnale subisce in questo modo attenuazioni ed amplificazioni diverse principalmente a seconda della direzione dalla quale proviene e della sua frequenza. Due esempi di riflessione dell'onda sono osservabili nella Figura 1.3. Il contributo della forma esterna degli orecchi acquista un rilievo molto maggiore nella situazione di *Near Field*, ovvero quando, come già detto in precedenza, la sorgente è posizionata a meno di un metro dal timpano e per la quale il fronte d'onda non può più essere considerato piano.

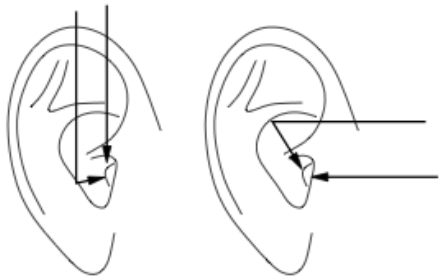

Figura 1.3: *Contributo dell'orecchio esterno*

La *Duplex Theory* dimostra inoltre come nella localizzazione verticale, ovvero nella determinazione della distanza dal suolo della sorgente (definito *angolo di elevazione* nella prossima sezione), sia la fisionomia soggettiva delle orecchie ad avere un peso maggiore rispetto agli altri fattori, mentre nella determinazione della direzione orizzontale (descritta da quello che verrà in seguito definito angolo *azimutale*), siano invece ITD e ILD ad avere maggior influenza.

1.1.2 Head Related Transfer Function

Gli effetti esaminati nella sezione precedente sono lineari, ciò significa che possono essere descritti da singole funzioni di trasferimento per poi essere convoluti. La pressione acustica prodotta da una sorgente al timpano pertanto è determinata univocamente da una risposta impulsiva che descrive la trasformazione che subisce il segnale sonoro dalla sorgente al timpano. Tale risposta impulsiva viene chiamata *Head Related Impulse Response* (hrir) e la sua trasformata di Laplace, chiamata *Head Related Transfer Function* (HRTF), è funzione delle tre coordinate spaziali e della frequenza ed ingloba tutti gli effetti fisici descritti nella sezione precedente.

Esistono più metodi per la determinazione e la gestione delle HRTF, i due principali sono i seguenti:

- HRTF ricavate sperimentalmente, ottenute mediante test che prevedono la registrazione delle risposte a determinati segnali di due microfoni posti nelle orecchie di un ascoltatore, che può essere una persona reale o un manichino.
- utilizzando i *modelli strutturali* che sono dati dalla composizione di singole funzioni di trasferimento relative alle principali componenti del corpo umano (testa, busto, spalle e orecchio esterno) e alla stanza.

Approssimando la testa con una sfera come nel modello proposto in precedenza, si possono sfruttare le coordinate sferiche (θ,ϕ,r) seguendo due approcci:

- Il *sistema di coordinate verticali polari*, (mostrato in Figura 1.4 a)), nel quale θ è rappresentato dall'angolo tra il piano yz ed il piano verticale contenente la sorgente e l'asse y, mentre ϕ è l'angolo a partire dal piano xy
- Il *sistema di coordinate interaurali polari*, (mostrato in Figura 1.4.b), nel quale ϕ è misurato come l'angolo tra il piano xy ed il piano contenente la sorgente e l'asse x, mentre θ è l'angolo dal piano yz.

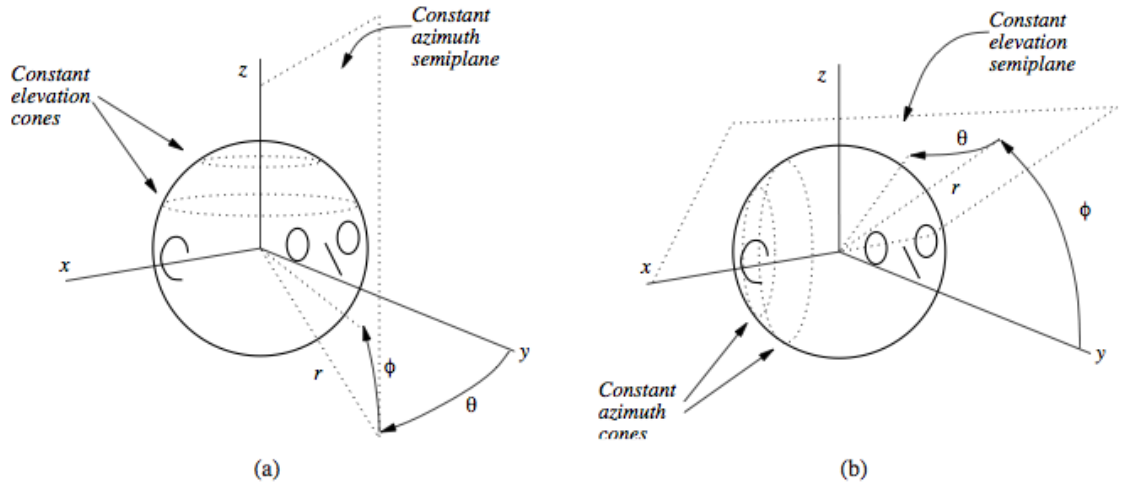

Figura 1.4: *Sistemi di coordinate sferiche: (a) verticali polari, (b) interaurali polari*

In entrambi θ rappresenta l'*angolo azimutale* (*azimuth*) e ϕ l'*angolo di elevazione*, mentre la coordinata radiale è definita *range* e viene indicata con *r*. È possibile osservare anche in figura 4 la presenza di regioni ad *azimuth* costante (che condividono quindi gli stessi valori di ITD e ILD) o ad *angolo di elevazione* costante.

Le *HRTF* vengono indicate con $H^{(l)(r)}(\theta, \phi, r, \omega)$ dove l ed r indicano la funzione di trasferimento relativa agli orecchi rispettivamente sinistro e destro. Quando r tende ad infinito si fa riferimento alla sopraccitata situazione di *far field*. Formalmente la funzione HRTF relativa ad un orecchio viene definita come il rapporto, dipendente dalla frequenza, tra il livello della pressione acustica al relativo timpano (*SPL-Sound Pressure Level*) e il livello della pressione acustica al centro della testa in condizioni di spazio libero (*free-field SPL*):

$$H^{(l)}(\theta, \phi, \omega) = \frac{SPL}{SPL_{free-field}} = \frac{\phi^{(l)}(\theta,\phi,\omega)}{\phi_f^{(l)},(\omega)}$$

$$H^{(r)}(\theta, \phi, \omega) = \frac{SPL}{SPL_{free-field}} = \frac{\phi^{(r)}(\theta,\phi,\omega)}{\phi_f^{(r)}(\omega)}$$

In Figura 1.5 è rappresentato il modello strutturale delle *HRTF* relative alle due orecchie, ponendo particolare attenzione su come queste comprendano gli effetti di tutti gli elementi fisici descritti finora.

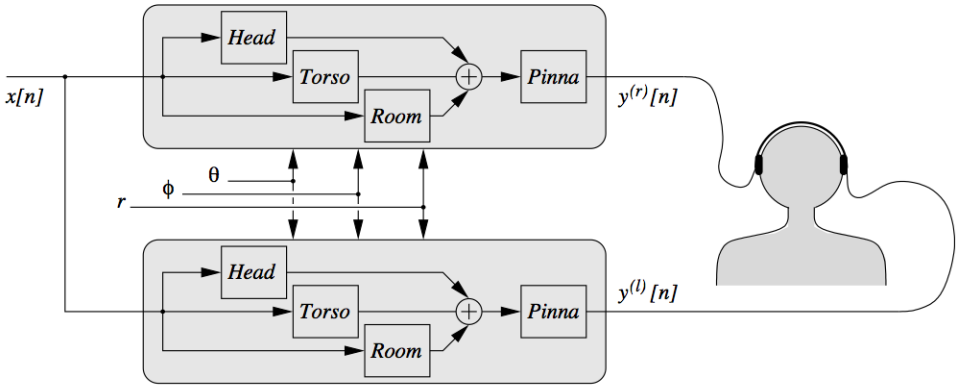

Figura 1.5: *Esempio di schema a blocchi di modello strutturale*

1.1.3 Localizzazione

Come già detto, l'informazione sull'angolo azimutale elaborata dal nostro apparato uditivo è determinata principalmente da ILD e ITD, ma può essere ambigua sia nel modello con testa sferica proposto che nella realtà.

Nel modello sferico una sorgente posizionata davanti all'ascoltatore ad un certo *azimuth* θ, produce gli stessi ILD ed ITD di una sorgente posta nel punto ad *azimuth* pari a π - θ. La conseguenza nell'ascolto dei due segnali appena descritti è la cosiddetta confusione *front-back*, secondo la quale l'ascoltatore non è in grado di distinguere i due segnali. Lo stesso concetto si estende ai cosiddetti *coni di confusione*, regioni dello spazio ad *azimuth* o *elevazione* costante che provocano gli effetti appena descritti. Nella realtà tali effetti si manifestano difficilmente

a causa della non sfericità della testa ed alla sua non simmetricità tra parte posteriore e parte anteriore.

Quando la spazializzazione del suono non avviene in modo corretto si parla di *lateralizzazione* e la percezione avviene come se la sorgente fosse posizionata all'interno della testa nell'asse che attraversa le due orecchie.

Con il termine *esternalizzazione* si indica invece la realizzazione di una sorgente simulata che viene percepita correttamente e localizzata all'esterno della testa. Un ruolo importante in questo caso è assunto dal riverbero ambientale, mentre nella determinazione della distanza della sorgente è l'intensità ad avere un peso fondamentale.

La realizzazione di *Virtual Auditory Scene* quindi è caratterizzata dall'applicazione delle teorie appena esposte e, come descritto fino ad ora, all'interno di esse la spazializzazione è data dalla differenza dei suoni percepiti ai due orecchi, che può essere realizzata in due modi:

- utilizzando l'ascolto con cuffie, situazione che permette il raggiungimento di livelli di interattività, realismo ed immersione molto maggiori
- mediante l'ascolto con altoparlanti, situazione più complicata in cui bisogna tenere conto che ogni orecchio sentirà tutte le fonti in gioco, le quali dovranno generare suoni studiati per interferire tra di loro in modo da far percepire ai timpani i segnali acustici desiderati.

1.2 Head Tracking

Il miglioramento della localizzazione di un suono può infine essere determinato da elementi dinamici: piccole variazioni della posizione della testa o della sorgente si traducono in grandi quantità di nuove informazioni relative alla sorgente che vengono processate in tempo reale dall'apparto uditivo umano. Il nostro sistema nervoso centrale è infatti in grado di confrontare i movimenti effettuati con le variazioni dei suoni percepiti ricavandone una determinazione più precisa della posizione della sorgente.

Per questo motivo l'interattività introdotta dall'head tracking permette un vantaggio notevole nella localizzazione dei suoni simulati, ricordando che con tale termine viene indicato il tracciamento della posizione della testa nel tempo. Tale tracciamento può avvenire principalmente in due modi: mediante strumenti di rilevamento di immagini, come ad esempio una web cam, oppure mediante sensori di movimento posizionati sopra alla testa dell'utente. È questa la strada scelta per il lavoro di tesi svolto. Un sensore fissato sulle cuffie indossate dall'ascoltatore ne comunicherà quindi l'orientazione al computer.

In accordo con il sistema di riferimento considerato in Figura 1.6 si definiscono:

- *Yaw*, chiamato anche "Heading", l'angolo di rotazione del sensore attorno all'asse z considerato in figura (coincidente con l'angolo azimutale citato in precedenza)
- *Pitch*, chiamato anche "Elevation", l'angolo di rotazione del sensore attorno all'asse x considerato in figura (coincidente con l'angolo di elevazione citato in precedenza)
- *Roll*, chiamato anche "Bank", l'angolo corrispondente al rollio della testa.

Corrispondenti ai cosiddetti angoli di Eulero.

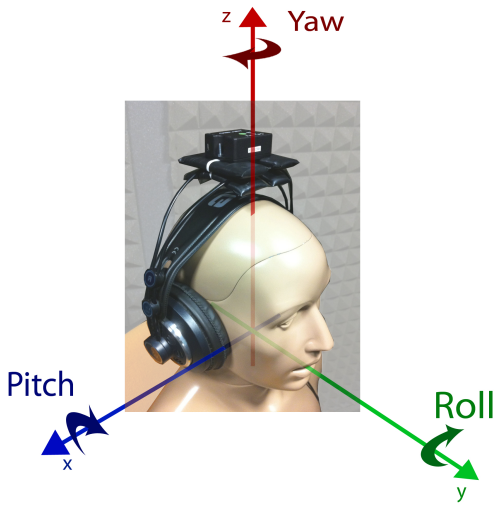

Figura 1.6: *Angoli di Eulero*

Capitolo 2

Sensore Trivisio Colibrì Wireless

2.1 Il sensore

Colibri-Wireless è il motion tracker d'avanguardia prodotto e distribuito dalla compagnia tedesca "Trivisio Prototyping GmbH" dotato di funzionalità Wireless. Il dispositivo è dotato di sensori a 3 assi in grado di misurare accelerazione, angolazione e campo magnetico (mediante un accelerometro, un giroscopio ed un magnetometro). Al suo interno è presente anche un sensore di temperatura che aiuta ad eliminare gli errori dovuti alle variazioni termiche che subiscono i sensori. Con la tecnologia wireless è possibile connettere fino a 10 sensori contemporaneamente alla stessa antenna USB. La frequenza di campionamento massima e di 100 Hz.

Trivisio mette a disposizione delle API per ambiente Windows e Linux che saranno approfondite in seguito.

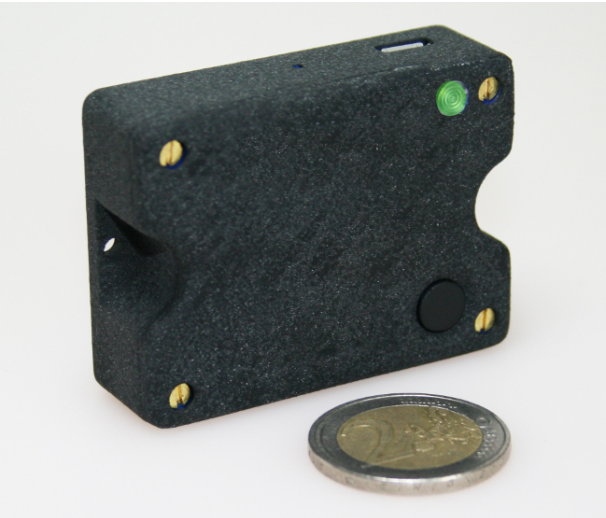

Figura 2.1: *Il sensore Trivisio Colibrì Wireless*

Queste le principali caratteristiche del sensore:

- Accelerometro MEMS a tre assi
- Giroscopio MEMS a tre assi
- Magnetometro AMR (magneto-resistivo) a tre assi
- Sensore di Temperatura
- Frequenza di operativit: 2,4 GHz
- Distanza massima supportata tra sensore e antenna: 10 metri
- Autonomia in modalit wireless di 16 ore, ricarica mediante USB
- Fino a 10 sensori connessi contemporaneamente alla stessa antenna
- Antenna USB in grado di connettersi a due o pi colibrì networks
- Ingressi ausiliari digitale ed analogico
- API in dotazione per Windows e Linux
- Accelerometro: Scala: $\pm$ 6g — Risoluzione: 13-bit
- Giroscopio: Scala: $\pm$ 2000° /s — Risoluzione: 16-bit
- Magnetometro: Scala: $\pm$ 1.3 Ga — Risoluzione: 12-bit
- Frequenza di campionamento: 100 Hz
- Precisione Orientamento: Pitch/roll: 0.5° — Yaw: 2°
- Sensore di Temperatura: Precisione: $\pm$ 0.5°C — Range: 0°C to +70°C
- 1024 bytes di memoria non volatile disponibili
- Alimentazione: 3.7V LiPo battery 660 mAh(40mA)
- Temperatura in funzionamento: 0 .. +55 °C (self-powered); 0 .. +40 °C (charging)
- Peso: 41g (with battery)
- Dimensioni (W/H/D): 56mm/42mm/17mm

Nella confezione vengono forniti un sensore Colibrì Wireless, una Antenna USB (USB-Dongle), un cavo microUSB-USB per il sensore ed un adattatore per la carica attraverso la rete elettrica domestica.

2.2 Installazione e funzionamento

2.2.1 Installazione

In ambiente Windows, per installare i Driver necessari alla comunicazione con il sensore è necessario svolgere le seguenti operazioni (File necessari reperibili allindirizzo `http://trivisio.com/index.php/support/software` o nel repository del Dipartimento di Ingegneria dell'Informazione dell'Università di Padova, Sound & Music Computing Group):

- Installare COLIBRI GUI SDK Version 2.1.1 per il sensore cablato:
  - File: Colibri-2.1.1-Windows-x86.exe
- Estrarre il contenuto dellarchivio SDK-Colibri_wireless_V3.0a3.zip:
  - Installare: Trivisio Colibri-3.0a3-win32.exe
  - Incollare i file:
    - dongleSetup.exe
    - SetRFChannel.exe

    nella cartella C:/Programs/Trivisio Colibri-3.0a3/Bin
- Copiare le seguenti dll presenti nella cartella C:/Programs/Trivisio Colibri-3.0a3/Bin:
  - Trivisio.dll
  - pthreadVC2.dll

  nella cartella Bin di Pure Data. (Ad esempio C:/Programmi/Pd/Bin)

Se questa operazione non viene eseguita l'external non può essere posizionata in Pure Data e il programma comunica la mancanza delle dll sopraccitate. Terminata l'installazione è possibile ad esempio utilizzare gli eseguibili in dotazione presenti nella cartella C:/Programs/Trivisio Colibri-3.0a3/Bin o sfruttare l'external progettata per Pure Data.

2.2.2 Funzionamento

Le principali operazioni possibili attraverso l'utilizzo dei driver proprietari sono elencate e descritte qui di seguito:

Connessione Antenna (Dongle)

L'Antenna viene connessa attraverso la porta USB. I driver necessari vengono installati automaticamente alla prima connessione se i software ColibriGUI+SDK e TrivisioColibrì sono stati installati correttamente.

Collegamento del Sensore

Come per l'antenna, i driver necessari vengono installati automaticamente alla prima connessione se i software ColibriGUI+SDK e TrivisioColibrì sono stati installati correttamente.

Accensione del Sensore

Se il sensore è spento l'unico modo per riaccenderlo è collegarlo al computer mediante il cavo USB in dotazione. Quando il cavo viene connesso il led sul dispositivo diventa arancione. Una volta acceso è possibile stabilire una connessione cablata oppure passare alla modalità wireless semplicemente rimuovendo il cavo USB. Il led sul dispositivo in questo caso diventa verde e segnala lo stato di ON del sensore.

Spegnimento del Sensore

Se il sensore è attivo in modalità wireless può essere spento utilizzando il cavo USB. Come nell'accensione appena il cavo viene collegato il led sul dispositivo si illumina di rosso per qualche istante. Per spegnere il dispositivo è necessario rimuovere il cavo prima che il led diventi arancione. Se l'operazione è stata eseguita con successo il led rimane spento. Se il sensore è connesso mediante USB invece è necessario scollegarlo (attivando la modalità wireless) e ripetere l'operazione appena descritta. Un secondo modo per spegnere il dispositivo consiste nell'usare l'eseguibile in dotazione ColibriTurnOff.exe o incapsulare le istruzioni necessarie come è stato fatto per l'esternal di PD progettata.

Sleep Mode e Wake Up Mode

Se il sensore è attivo e scollegato, entra in Sleep Mode dopo un tempo prefissato che dipende dalla modalità attiva in quel momento. Tale modalità è impostabile mediante l'external progettata e può essere settata nei seguenti modi:

- Mode 0: Immediatamente
- Mode 1: 1 Minuto
- Mode 2: 2 Minuti
- Mode 3: 5 Minuti
- Mode 4: 10 Minuti

A seconda invece della modalità di Wake Up attiva, il sensore può essere "risvegliato" nei seguenti modi:

- Mode 0: Solamente mediante connessione Cavo USB
- Mode 1: Mediante cavo USB o muovendo il sensore con uno shake delicato
- Mode 2: Mediante connessione wireless. In questo caso rimane attivo solo il modulo radio del dispositivo, riducendo il consumo di batteria ad 1/3 rispetto al normale funzionamento.

Di fatto se il sensore entra in Sleep Mode quando è impostato il Wake Up Mode 0, il dispositivo risulta essere spento. (Combinazione utilizzata nel metodo turn_off).

Carica del Sensore

La carica del sensore avviene collegando il cavo USB al computer o alla rete domestica utilizzando l'adattatore fornito nella confezione. Quando il sensore è in carica il led sul dispositivo si illumina di arancione, quando la carica è completa il led diventa verde.

2.3 Esempi

Gli eseguibili forniti vanno lanciati da riga di comando DOS in modo da poter comunicare eventuali opzioni aggiuntive e poter visualizzare gli output. Di seguito i principali esempi:

dongleSetup.exe

Il sensore e l'antenna USB (USB-Dongle) possono comunicare tra loro se sono settati sullo stesso canale radio (RF-Channel). Anche il sensore deve essere collegato all'antenna mediante il suo numero seriale. Fino a 10 sensori wireless possono essere collegati allo stesso canale radio. Alla consegna e al reset delle impostazioni il canale impostato sul sensore e sull'antenna è lo stesso (Channel 25) e tra loro risultano già associati. Per modificare le associazioni e lo stato di attivazione può essere usato l'eseguibile fornito in dotazione dongleSetup.exe.

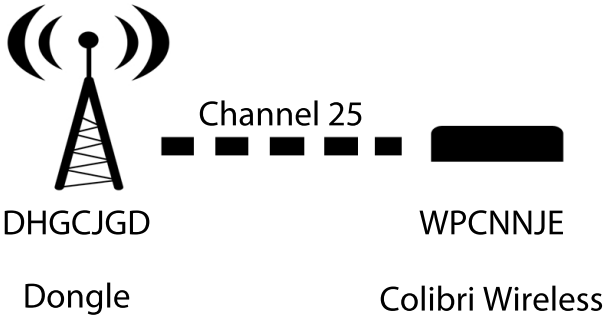

Figura 2.2: *Esempio di connessione*

Segue un esempio di ciò che si vede nella finestra DOS al lancio dell'eseguibile relativamente alla colibrì network mostrata in Figura 2.2:

```
Colibri-WD ID: DHGCJGD
=======================
Radio channel: 25
Pos       ID        Active
a         WPCNNJE   true
b                   false
c                   false
d                   false
e                   false
f                   false
g                   false
h                   false
i                   false
j                   false
L(ist),I(d setting),T(oggle),R(adio channel),S(ave),Q(uit) >
```

Vengono quindi comunicati il numero seriale dell'antenna (in questo esempio DHGCJGD), il Canale Radio utilizzato (in questo esempio Canale 25), il numero seriale dei sensori connessi ed il loro stato di attivazione (in questo caso WPCNNJE unico sensore collegato e attivo).

Sono inoltre disponibili le seguenti funzioni richiamabili digitando la loro lettere iniziale.
L(ist): Per mostrare la lista dei sensori connessi
I(d setting): per connettere il sensore con il suo numero seriale
T(oggle): Per attivare o disattivare il sensore
R(adio channel): per cambiare il canale radio utilizzato dallantenna
S(ave): Per salvare le impostazioni attuali
Q(uit): Per uscire dal dongleSetup.

SetRFChannel.exe

Per cambiare il canale radio utilizzato da un sensore è necessario lanciare tale eseguibile dopo aver connesso l'antenna e aver attivato il sensore wireless. Dopo l'avvio, premere "Refresh Dongle" e il programma mostra il seriale dellantenna e dei sensori ad essa associati. Il canale radio in uso è visibile nel campo "Channel". Per cambiarlo basta inserire il nuovo canale radio nel campo "Channel" e premere il pulsante "Set" e seguire le istruzioni del software.

Multiorientation.exe

Multiorientation permette di visualizzare un rendering grafico 3D della posizione attuale del sensore connesso. Lanciare l'eseguibile da riga di comando DOS e aggiungere "-1" alla fine se si sta tentando di connettere un sensore wireless. (Es. riga di comando Dos: "C:/Program Files/Trivisio Colibri-3.0a3/Bin> multiorientation.exe -1"). Se la connessione va a buon fine si apre una finestra in cui è visibile il sensore in un grafico 3D utilizzando un rendering video attraverso OpenGL e glut. A questo punto possibile:

- premere "q" per uscire dal programma e interrompere la connessione.
- Premere "a" per resettare i valori di Yaw, Pitch e Roll secondo l"attuale posizione del sensore,
- Premere "h" per resettare solamente il dato di Yaw,
- Premere "o" per i soli dati di Pitch e Roll,
- Premere "r" per tornare all'allineamento iniziale e
- Premere "j" per attivare/disattivare la funzione di "jitter reduction" descritta successivamente.

Per ottenere dati da sensori multipli per l'esportazione in Excel o Matlab è possibile usare ColibriMultiCollect.exe.

ColibriWCalibMag.exe

Anche se il sensore colibrì è stato testato e calibrato in fase di produzione, poiché la calibrazione magnetica è molto sensibile alle influenze esterne (in particolar modo le interferenze metalliche ed elettromagnetiche di altri dispositivi nelle vicinanze danneggiano il tracciamento dei risultati), può rendersi necessaria la ricalibrazione nellambiente e nelle condizioni in cui verrà poi utilizzato per poter ottenere risultati migliori. Per la calibrazione è necessario lanciare ColibriWCalibMag.exe da riga di comando DOS aggiungendo dopo uno spazio il seriale del sensore sul quale verrà effettuata l'operazione. (Es. "C:/Program Files/Trivisio Colibri-3.0a3/Bin> ColibriWCalibMag.exe WPCNNJE"). Dopo circa 3 secondi il software comincia le misurazioni per la ricalibrazione del magnetometro. Durante queste misurazioni il sensore deve essere ruotato lentamente attorno ognuno dei suoi tre assi (consigliabile ruotare il dispositivo su un piano orizzontale). Dopo aver effettuato 3000 misure nei successivi 50 secondi il software conclude la ricalibrazione. Se l'operazione è andata a buon fine l'errore nella misura dellangolo di Yaw deve essere di circa +/- 2° nelle rotazioni di 90°. Se il risultato è peggiore, ripetere la ricalibrazione. In ogni caso è meglio evitare la presenza di grossi oggetti di ferro nelle vicinanze del sensore.

ColibriFactoryReset.exe

Permette il ripristino delle impostazioni di fabbrica. Il sensore deve essere connesso mediante cavo USB. L'eseguibile deve essere chiamato da riga di comando DOS con l'aggunta di "s" dopo uno spazio altrimenti il reset non viene salvato. (Es. "C:/Program Files/Trivisio Colibri-3.0a3/Bin> ColibriFactoryReset.exe -s").

ColibriUpgradeFW.exe

Per eseguire l'aggiornamento del firmware del sensore è necessario scaricare l'ultimo firmware all'indirizzo `http://trivisio.com/index.php/support/software`. Verificare la versione attuale per evitare rischiosi downgrade. Connettere il sensore via cavo e lanciare l'eseguibile in questione da riga di comando DOS. Vengono visualizzati i sensori connessi e il loro attuale firmware in un elenco numerato. Digitare il numero del sensore da aggiornare e premere invio. Dopo aver letto le indicazioni che compaiono, digitare "y" e premere invio. Si apre una finestra "FIRMWARE" nella quale incollare il file di firmware scaricato in precedenza (es. colibri1409.fmw). Dopo aver chiuso tale finestra, scollegare il sensore per un secondo e ricollegarlo per completare l'update. L'aggiornamento del firmware dell'antenna si esegue allo stesso modo utilizzando il file ColibriUpgradeFW.exe.

2.4 API

Per una descrizione pi ù dettagliata si faccia riferimento al file UserManual_Colibri_API.pdf relativo alla versione 3.0a3, contenuto nellarchivio SDK-Colibri_wireless_V3.0a3.zip citato in precedenza. Gli header file disponibili nelle API sono: TrivisioTypes.h, TrivisoColibri.h, TrivisioWirelessColibri.h e TrivisioConfig.h.

2.4.1 TrivisioTypes.h - Data Structures

Le Data Structures supportate dalle API e definite nel file *TrivisioTypes.h* sono le seguenti:

- ColibriConfig
- DongleConfig
- TrivisioDongleSensorEntry
- TrivisioIMUCalibration
- TrivisioIMUData
- TrivisioSensor.

ColibriConfig

Un dato di tipo *ColibriConfig* descrive le impostazioni del sensore colibri, in particolare contiene i campi mostrati nella Tabella 2.1.

ascii	ASCII output ON/OFF
autoStart	AutoStart on Turn On
freq	Frequenza di campionamento dati
magDiv	Il divisore del magnetometro in caso di connessione a colibrì via cavo
magGain	Il guadagno del magnetometro in caso di connessione a colibrì wireless
raw	Raw Data Switch
CalibriConfig::Sensor	Classe interna che contiene le costanti dei sensori di cui dispone il colibrì relative ai tre assi del magnetometro, dei due accelerometri, del giroscopio, della temperatura e dell'orientazione
sensor	Dato di tipo sensor definito in questa struttura, contenente le costanti dei sensori

Tabella 2.1: *Campi ColibriConfig*

DongleConfig

Un dato di tipo *DongleConfig* descrive le impostazioni dell'antenna wireless, in particolare contiene i campi mostrati nella Tabella 2.2.

ascii	ASCII output ON/OFF
autoStart	AutoStart on Turn On
count	Counter on Timestamp: 1 o 0
freq	Frequenza di campionamento
radioChannel	Canale radio utilizzato, parametro configurabile con gli eseguibili citati nel paragrafo precedente

Tabella 2.2: *Campi DongleConfig*

DongleSensorEntry

Un dato di tipo *DongleSensorEntry* descrive un singolo sensore presente nella rete che l'antenna sta tentando di inizializzare, in particolare contiene i campi mostrati nella Tabella 2.3.

ID	identificativo del sensore, il seriale
active	lo stato di attivazione del sensore
pos	la posizione nella lista di sensori rilevati

Tabella 2.3: *Campi DongleSensorEntry*

TrivisioIMUCalibration

Un dato di tipo *TrivisioIMUCalibration* descrive i parametri utilizzati nella calibrazione di un sensore colibri, i campi contengono i fattori di scala e i valori di compensazione di allineamento degli accelerometri, del magnetometro e del giroscopio relativi ad ogni asse. Come detto in precedenza per risultati migliori è utile ricalibrare il sensore nelle condizioni in cui sarà usato e lo si può fare utilizzando l'eseguibile in dotazione già citato. I campi contenuti sono i seguenti: ba[3], ba2[3], bg[3], bm[3], ka[3], ka2[3], kg[3], Kga[9], km[3], Ra[9], Ra2[9], Rg[9], Rm[9].

TrivisioIMUData

Un dato di tipo *TrivisioIMUData* contiene tutti i dati rilevati dai sensori presenti nel colibrì in un preciso istante, in particolare figurano i campi mostrati nella Tabella 2.4.

acc_x, acc_y, acc_z,	Valori relativi ai 3 assi
acc2_x, acc2_y, acc2_z,	rilevati dai due accelerometri,
gyr_x, gyr_y, gyr_z,	dal giroscopio
mag_x, mag_y, mag_z	e dal magnetometro
q_w, q_x, q_y, q_z	le quattro componenti relative
	all'orientazione dei quaternioni)
t	Istante di tempo a cui si riferiscono i dati)
temp	Temperatura rilevata dal sensore interno al colibri

Tabella 2.4: *Campi TrivisioIMUData*

TrivisioSensor

Un dato di tipo *TrivisioSensor* descrive un sensore e le modalità di accesso ad esso. Può essere usato anche per rappresentare un'antenna. In particolare contiene i campi mostrati nella Tabella 2.5.

ID	Identificativo del sensore, il seriale)
dev	Nome del dispositivo se presente)
FWsubver	Firmware subversion)
FWver	Firmware version)
HWver	Hardware version)
Type	Dato di tipo SensorType definito in questa struttura che può essere
	COLIBRI, COLIBRI_W, COLIBRI_WD).

Tabella 2.5: *Campi TrivisioSensor*

2.4.2 TrivisioColibri.h - Metodi di Base

Queste le funzioni supportate e definite nel file *TrivisioColibri.h*:

*Trivisio_DECLSPEC const char * colibriAPIVersion ()*
Fornisce la versione delle API in uso.

*Trivisio_DECLSPEC void colibriBoresight (void * imu, enum ColibriBoresightType type)*
Funzione utilizzata per il riallineamento del sensore relativo all'handle `imu` secondo la posizione attuale. Il tipo *ColibriBoresightType* è definito in *TrivisioTypes.h* e può essere COLIBRI_HEADING_RESET, COLIBRI_OBJECT_RESET e COLIBRI_ALIGNMENT_RESET. Le conseguenze sui valori di Yaw, Pitch e Roll calcolate sul dato di tipo *TrivisioIMUData* sono le seguenti:
COLIBRI_ALIGNMENT_RESET azzera tutti e tre i valori.
COLIBRI_HEADING_RESET azzera il valore di Yaw lasciando invariati gli altri due.
COLIBRI_OBJECT_RESET azzera i valori di Pitch e Roll lasciando invariato il valore di Yaw.

*Trivisio_DECLSPEC void colibriClose (void * imu)*
Chiude la connessione aperta tra l'handle `imu` e il sensore fisico.

*Trivisio_DECLSPEC void * colibriCreate (unsigned short bufLen)*
Crea un nuovo Colibrì handle utilizzato per comunicare con il sensore fisico. Il parametro trasmesso `bufLen` indica la lunghezza del buffer interno di output che deve essere utilizzato. Un buffer corto (0) assicura il ritardo minimo tra la misura e la disponibilità dei dati misurati a costo di una probabile perdita di misurazioni Un buffer lungo potrebbe necessitare l'attesa del dato prima che esso sia disponibile.

*Trivisio_DECLSPEC void colibriDestroy (void * imu)*
Elimina l'handle `imu` creato in precedenza.

*Trivisio_DECLSPEC voidcolibriEulerOri (const struct TrivisioIMUData * data, float euler[3])*
Converte i dati misurati contenuti nella variabile `data` di tipo *TrivisioIMUData* nei tre angoli di Eulero corrispondenti all'orientazione del sensore, resituendoli nel vettore `euler` passato per parametro.

*Trivisio_DECLSPEC int colibriGetBoresight (const void * imu)*
Per ottenere lo stato dellallineamento del sensore relativo all'handle `imu`. Restituisce nonzero se l'allineamento è attivo.

*Trivisio_DECLSPEC voidcolibriGetCalib (const void * imu, struct TrivisioIMUCalibration * calib)*
Comunica un dato di tipo *TrivisioIMUCalibration* che descrive calibrazione in uso nel sensore relativo all'handle `imu` nella variabile `calib` passata come parametro.

*Trivisio_DECLSPEC void colibriGetConfig (const void * imu, struct ColibriConfig * config)*
Comunica un dato di tipo *ColibriConfig* che descrive le impostazioni in uso nel sensore relativo all'handle `imu` nella variabile `config` passata come parametro.

*Trivisio_DECLSPEC void colibriGetData (void * imu, struct TrivisioIMUData * data)*
Comunica un dato di tipo *TrivisioIMUData* contentente gli ultimi dati misurati dal sensore relativo all'handle `imu` nella variabile `data` passata come parametro. Come spiegato in precedenza, un dato di questo tipo contiene un campo relativo all'istante di rilevamento . Tale campo è uguale a zero se il dato è stato trasmesso a dispositivo "stoppato".

*Trivisio_DECLSPEC int colibriGetDeviceList (struct TrivisioSensor * list, unsigned listLen)*
Utilizzata per rilevare fino ad un numero pari a `listLen` di sensori disponibili. `list` costituisce la lista (di dati di tipo *TrivisioSensor*) di sensori rilevati. Restituisce il numero minore uguale a `listLen` di sensori rilevati oppure "-1" se eccede `listLen`.

*Trivisio_DECLSPEC void colibriGetID (const void * imu, char * ID)*
Comunica l'ID del sensore relativo all'handle `imu` nella variabile `ID` passata come parametro.

*Trivisio_DECLSPEC void colibriGetJitterParam (const void * imu, float * lambda, float * th, float * tol)*
Comunica i parametri usati per la"jitter reduction" (riduzione delle variazioni indesiderate dei valori misurati) del sensore relativo all'handle `imu` nelle variabili passate come parametro.

*Trivisio_DECLSPEC int colibriGetJitterStatus (const void * imu)*
Restituisce 0 se la "jitter reduction" del sensore relativo all'handle `imu` è disattiva.

*Trivisio_DECLSPEC void colibriGetKa (const void * imu, float Ka[9])*
Comunica i valori di guadagno dei filtri per le misurazioni dell"accelerometro del sensore relativo all'handle`imu` nella variabile `Ka` passata come parametro.

*Trivisio_DECLSPEC int colibriGetKaStatus (const void * imu)*
Restituisce lo stato del filtro Kalman sui dati dell'accelerometro del sensore relativo all'handle `imu`: zero se non attivo.

*Trivisio_DECLSPEC void colibriGetKg (const void * imu, float Kg[9])*
Comunica i valori di guadagno dei filtri per le misurazioni del giroscopio del sensore relativo all'handle `imu` nella variabile `Kg` passata come parametro.

*Trivisio_DECLSPEC int colibriGetKgStatus (const void * imu)*
Restituisce lo stato del filtro Kalman sui dati del giroscopio del sensore relativo all'handle `imu`: zero se non attivo.

*Trivisio_DECLSPEC void colibriGetKm (const void * imu, float Km[9])*

Comunica i valori di guadagno dei filtri per le misurazioni del magnetometro del sensore relativo all'handle`imu` nella variabile `Kg` passata come parametro.

*Trivisio_DECLSPEC int colibriGetKmStatus (const void * imu)*

Restituisce lo stato del filtro Kalman sui dati del magnetometro del sensore relativo all'handle `imu`: zero se non attivo.

*Trivisio_DECLSPEC void colibriGetSensorInfo (const void * imu, struct TrivisioSensor * info)*

Comunica i dati del sensore relativo all'handle `imu` attraverso la variabile `info` di tipo *TrivisioSensor* passata come parametro.

*Trivisio_DECLSPEC uint8_t colibriGetTriggerDivisor (const void * imu)*

Restituisce il valore del "trigger divisor" del sensore relativo all'handle `imu`. 0 se non è emesso alcun segnale di trigger.

*Trivisio_DECLSPEC int colibriOpen (void * imu, const struct ColibriConfig * conf, const char * dev)*

Apre la connessione tra l'handle `imu` e il sensore fisico `dev` utilizzando le impostazioni contenute in `conf`. Restituisce 0 in caso di successo.

*Trivisio_DECLSPEC int colibriSaveSettings (void * imu)*

Salva le attuali impostazioni in modo permanente nel sensore relativo all'handle `imu`. Restituisce 0 in caso di successo.

*Trivisio_DECLSPEC void colibriSetBoresight (void * imu, int active)*

Attiva l'allineamento del sensore relativo all'handle `imu` se il parametro `active` è diverso da zero, altrimenti lo disattiva.

*Trivisio_DECLSPEC void colibriSetCalib(void?imu, const struct TrivisioIMUCalibration * calib)*

Imposta la calibrazione del sensore relativo all'handle `imu` secondo i dati contenuti nella variabile `calib` (di tipo *TrivisioIMUCalibration*) passata per parametro.

*Trivisio_DECLSPEC void colibriSetConfig(void * imu, const struct ColibriConfig * config)*

Setta le impostazioni in uso nel sensore relativo all'handle `imu` secondo i dati contenuti nella variabile `config` (di tipo *ColibriConfig*) passata per parametro.

*Trivisio_DECLSPEC void colibriSetJitterParam (void * imu, float lambda, float th, float tol)*

Imposta i parametri usati per la "jitter reduction" (riduzione delle variazioni indesiderate dei valori misurati) del sensore relativo all'handle `imu` secondo i dati passati per parametro.

*Trivisio_DECLSPEC void colibriSetJitterStatus (void * imu, int status)*

Attiva la "jitter reduction" del sensore relativo all'handle `imu` se il parametro `status` è diverso da zero, altrimenti la disattiva.

*Trivisio_DECLSPEC void colibriSetKa (void * imu, const float Ka[9])*

Imposta i valori di guadagno dei filtri per le misurazioni dell'accelerometro del sensore relativo all'handle `imu` secondo i dati contenuti variabile `Ka` passata come parametro.

*Trivisio_DECLSPEC void colibriSetKaStatus (void * imu, int active)*

Attiva il filtro Kalman sui dati dell'accelerometro del sensore relativo all'handle `imu` se il parametro `active` è diverso da zero, altrimenti lo disattiva.

*Trivisio_DECLSPEC void colibriSetKg (void * imu, const float Kg[9])*

Imposta i valori di guadagno dei filtri per le misurazioni del giroscopio del sensore relativo all'handle `imu` secondo i dati contenuti nella variabile `Kg` passata come parametro.

*Trivisio_DECLSPEC void colibriSetKgStatus (void * imu, int active)*

Attiva il filtro Kalman sui dati del giroscopio del sensore relativo all'handle `imu` se il parametro `active` è diverso da zero, altrimenti lo disattiva.

*Trivisio_DECLSPEC void colibriSetKm (void * imu, const float Km[9])*

Imposta i valori di guadagno dei filtri per le misurazioni del magnetometro del sensore relativo all'handle `imu` secondo i dati contenuti nella variabile `Km` passata come parametro.

*Trivisio_DECLSPEC void colibriSetKmStatus (void * imu, int active)*

Attiva il filtro Kalman sui dati del magnetometro del sensore relativo all'handle `imu` se il parametro `active` è diverso da zero, altrimenti lo disattiva.

*Trivisio_DECLSPEC void colibriSetTriggerDivisor (void * imu, uint8_t div)*

Imposta il valore del "trigger divisor" del sensore relativo all'handle `imu`.

*Trivisio_DECLSPEC int colibriStart (void * imu)*

Avvia le misurazioni del sensore relativo all'handle `imu`. Restituisce 1 in caso di successo.

*Trivisio_DECLSPEC void colibriStop (void * imu)*

Ferma le misurazioni del sensore relativo all'handle `imu`.

2.4.3 TrivisioWirelessColibri.h - Metodi Colibrì Wireless

Di seguito le funzioni definite nel file *TrivisioWirelessColibri.h*:

Trivisio_DECLSPEC float colibriCalcWirelessChargeRatio (float voltage)

Restituisce la percentuale di batteria residua relativa alla tensione passata per parametro.

*Trivisio_DECLSPEC void colibriDongleClose (void * dongle)*

Chiude la connessione tra la dongle-handle `dongle` e l'antenna fisica.

*Trivisio_DECLSPEC void * colibriDongleCreate ()*

Crea una nuova dongle-handle.

*Trivisio_DECLSPEC void colibriDongleDestroy (void * dongle)*

Elimina la dongle-handle `dongle` creata in precedenza.

*Trivisio_DECLSPEC int colibriDongleDiscover (void * dongle, struct TrivisioDongleSensorEntry * list, unsigned listLen)*

Utilizzata per rilevare fino ad un numero pari a `listLen` di sensori disponibili (attivi o non attivi) nel canale radio impostato nell'antenna relativa alla dongle-handle `dongle`. `list` costituisce la lista (di dati di tipo *TrivisioDongleSensorEntry*) di sensori wireless rilevati. Restituisce il numero minore uguale a `listLen` di sensori rilevate oppure "-1" se eccede `listLen`.

*Trivisio_DECLSPEC int colibriDongleGetActiveSensorList (const void * dongle, struct TrivisioDongleSensorEntry * list, unsigned listLen)*

Utilizzata per rilevare fino ad un numero pari a `listLen` di sensori disponibili e attivi nel canale radio impostato nell'antenna relativa alla dongle-handle `dongle`. `list` costituisce la lista (di dati di tipo *TrivisioDongleSensorEntry*) di sensori wireless rilevati. Restituisce il numero minore uguale a `listLen` di sensori rilevate oppure "-1" se eccede `listLen`.

*Trivisio_DECLSPEC void colibriDongleGetConfig (const void * dongle, struct DongleConfig * config)*

Comunica un dato di tipo `DongleConfig` che descrive le impostazioni in uso nell'antenna relativa alla dongle-handle `dongle` nella variabile `config` passata come parametro.

*Trivisio_DECLSPEC int colibriDongleGetDeviceList (struct TrivisioSensor * list, unsigned listLen)*

Utilizzata per rilevare fino ad un numero pari a `listLen` di antenne disponibili. `list` costituisce la lista (di dati di tipo *TrivisioSensor*) di antenne rilevate. Restituisce il numero minore uguale a `listLen` di antenne rilevati oppure "-1" se eccede `listLen`.

*Trivisio_DECLSPEC void colibriDongleGetID (const void * dongle, char * ID)*

Comunica l'ID dell'antenna relativa al dongle-handle `dongle` nella variabile `ID` passata come parametro.

*Trivisio_DECLSPEC struct TrivisioDongleSensorEntry colibriDongleGetSensor (const void * dongle, uint8_t pos)*

Restituisce un dato di tipo *TrivisioDongleSensorEntry* relativo al sensore in posizione `pos` della rete wireless gestita dalla dongle-handle `dongle`.

*Trivisio_DECLSPEC void colibriDongleGetSensorInfo (const void * dongle, struct TrivisioSensor * info)*

Comunica i dati dell'antenna relativa alla dongle-handle `dongle` attraverso la variabile `info` di tipo `TrivisioSensor` passata come parametro.

*Trivisio_DECLSPEC int colibriDongleGetSensorList (const void * dongle, struct TrivisioDongleSensorEntry * list, unsigned listLen)*

Utilizzata per ottenere la lista di sensori (di lunghezza `listLen`) nella rete gestita dall'antenna relativa alla dongle-handle `dongle`. `list` costituisce la lista (di dati di tipo *TrivisioDongleSensorEntry*) di sensori wireless rilevati. Restituisce il numero minore uguale a `listLen` di sensori rilevate oppure "-1" se eccede `listLen`.

*Trivisio_DECLSPEC int colibriDongleOpen (void * dongle, const struct DongleConfig * conf, const char * dev)*

Apre la connessione tra la dongle-handle `dongle` e l'antenna fisica `dev` utilizzando le impostazioni contenute in `conf`. Restituisce 0 in caso di successo.

*Trivisio_DECLSPEC int colibriDongleSaveSettings (void * dongle)*

Salva le attuali impostazioni in modo permanente nell'antenna relativa alla dongle-handle `dongle`. Restituisce 0 in caso di successo.

*Trivisio_DECLSPEC void colibriDongleSetConfig (void * dongle, const struct DongleConfig * config)*

Setta le impostazioni in uso nell'antenna relativa alla dongle-handle `dongle` secondo i dati contenuti nella variabile `config` (di tipo *ColibriConfig*) passata per parametro.

*Trivisio_DECLSPEC void colibriDongleSetSensorList (void * dongle, const struct TrivisioDongleSensorEntry * list, unsigned listLen)*

Utilizzata per impostare la lista (lunga `listLen`) di sensori gestita dall'antenna relativa alla dongle-handle `dongle` attraverso la lista `list` di variabili di tipo *TrivisoDongleSensorEntry*.

*Trivisio_DECLSPEC int colibriDongleStart (void * dongle)*

Avvia la trasmissione dati di tutti i sensori presenti nella rete gestita dall'antenna relativa alla dongle-handle `dongle`.

*Trivisio_DECLSPEC void colibriDongleStop (void * dongle)*

Ferma la trasmissione dati di tutti i sensori presenti nella rete gestita dall'antenna relativa alla dongle-handle `dongle`. I sensori continuano comunque ad effettuare le misurazioni.

*Trivisio_DECLSPEC float colibriGetWirelessBatteryLevel (const void * imu)*

Restituisce la tensione della batteria del sensore relativo all'handle `imu`.

*Trivisio_DECLSPEC void colibriGetWirelessKa2 (const void * imu, float Ka2[9])*

Comunica i valori di guadagno dei filtri per le misurazioni del secondo accelerometro del sensore relativo all'handle `imu` nella variabile `Ka2` passata come parametro.

*Trivisio_DECLSPEC int colibriGetWirelessKa2Status (const void * imu)*

Restituisce lo stato del filtro Kalman sui dati del secondo accelerometro del sensore relativo all'handle `imu`: zero se non attivo.

*Trivisio_DECLSPEC uint8_t colibriGetWirelessRadioChannel (const void * imu)*

Restituisce il canale radio utilizzato dal sensore relativo all'handle `imu`.

*Trivisio_DECLSPEC uint8_t colibriGetWirelessSleepTimeOut (const void * imu)*

Restituisce la modalità di sleep time out in uso nel sensore relativo all'handle `imu`.
Mode 0: il sensore entra in sleep mode immediatamente.
Mode 1: il sensore entra in sleep mode dopo 1 minuto.
Mode 2: il sensore entra in sleep mode dopo 2 minuti.
Mode 3: il sensore entra in sleep mode dopo 5 minuti.
Mode 4: il sensore entra in sleep mode dopo 10 minuti.

*Trivisio_DECLSPEC uint8_t colibriGetWirelessWakeUpMode (const void * imu)*

Restituisce la modalità di wake up in uso nel sensore relativo all'handle `imu`.
Mode 0: il sensore può uscire dallo sleep mode solo venendo collegato al computer mediante usb.
Mode 1: il sensore può uscire dallo sleep mode venendo collegato al computer mediante usb o venendo mosso delicatamente.
Mode 2: il sensore può uscire dallo sleep mode mediante connessione wireless. In questo modo solo il modulo radio del sensore rimane attivo ed il consumo viene ridotto ad un terzo.

*Trivisio_DECLSPEC int colibriOpenWireless (void * imu, void * dongle, const struct ColibriConfig * conf, uint8_t pos)*

Apre la connessione tra l'antenna relativa alla dongle-handle `dongle` e il sensore relativo all'handle `imu` utilizzando le impostazioni contenute in `conf`.

*Trivisio_DECLSPEC void colibriSetWirelessKa2 (void * imu, const float Ka2[9])*

Imposta i valori di guadagno dei filtri per le misurazioni del secondo accelerometro del sensore relativo all'handle `imu` secondo i dati contenuti nella variabile `Ka2` passata come parametro.

*Trivisio_DECLSPEC void colibriSetWirelessKa2Status (void * imu, int active)*

Attiva il filtro Kalman sui dati del secondo accelerometro del sensore relativo all'handle `imu` se il parametro `active` è diverso da zero, altrimenti lo disattiva.

*Trivisio_DECLSPEC void colibriSetWirelessRadioChannel (void * imu, uint8_t channel)*

Imposta il canale radio utilizzato dal sensore relativo all'handle `imu` come descritto da `channel`.

*Trivisio_DECLSPEC void colibriSetWirelessSleepTimeOut (void * imu, uint8_t timeOut)*

Imposta la modalità di sleep time out in uso nel sensore relativo all'handle `imu` come descritto da `timeout`.
Mode 0: il sensore entra in sleep mode immediatamente.
Mode 1: il sensore entra in sleep mode dopo 1 minuto.
Mode 2: il sensore entra in sleep mode dopo 2 minuti.
Mode 3: il sensore entra in sleep mode dopo 5 minuti.
Mode 4: il sensore entra in sleep mode dopo 10 minuti.

*Trivisio_DECLSPEC void colibriSetWirelessWakeUpMode (void * imu, uint8_t mode)*

Imposta la modalità di wake up in uso nel sensore relativo all'handle `imu` come descritto da `mode`.
Mode 0: il sensore può uscire dallo sleep mode solamente venendo collegato al computer mediante usb.
Mode 1: il sensore può uscire dallo sleep mode venendo collegato al computer mediante usb o venendo mosso delicatamente.
Mode 2: il sensore può uscire dallo sleep mode mediante connessione wireless. In questo modo solo il modulo radio del sensore rimane attivo ed il consumo viene ridotto ad un terzo.

2.4.4 Come stabilire una connessione cablata

Per stabilire una connessione con un sensore è necessario prima creare un handle (che di fatto rappresenta un sensore, in seguito nominato imu) mediante la funzione *colibriCreate()* ed in seguito aprire la connessione tra l'handle e il sensore fisico mediante la funzione *colibriOpen()*. Per conoscere quali dispositivi sono disponibili è fornita la funzione *colibriGetDeviceList()*. A questo punto è possibile accedere al sensore ed utilizzare tutte le funzioni fornite nelle API. Per avviare e fermare le misurazioni sono necessarie le funzioni *colibriStart()* e *colibriStop()*. Mentre le misurazioni sono in corso non è possibile utilizzare funzioni che variano le impostazioni del sensore, eccetto quelle di riallineamento e di accesso ai dati misurati. La connessione viene chiusa mediante la funzione *colibriClose()*. A questo punto è possibile eliminare lhandle mediante *colibriDestroy()*. A tali operazioni ne vanno aggiunte altre, analizzate nella prossima sezione, in caso di connessione wireless. Segue un esempio di codice C++ utilizzato per stabilire una connessione cablata (restituisce true in caso di successo):

```
void * imu;
TrivisioSensor sensorList[10];
if (!(sensors = colibriGetDeviceList(sensorList, 10)))
   return false;
imu = colibriCreate(0);
if (colibriOpen(imu, 0, sensorList[0].dev)) return false;
// connessione aperta
colibriStart(imu);
return sensors!=0
// connessione stabilita
colibriStart(imu);
// misurazioni avviate, possibile accedere ai dati
colibriStop(imu);
// misurazioni stoppate
colibriClose(imu);
// connessione chiusa
colibriDestroy(imu);
//variabili eliminate
```

2.4.5 Come stabilire una connessione Wireless

Per stabilire una connessione con un sensore wireless è necessario prima creare una dongle-handle (che di fatto rappresenta un'antenna wireless) mediante la funzione *dongleCreate()* ed in seguito aprire la connessione tra la dongle-handle e l'antenna fisica mediante la funzione *colibriDongleOpen()*. Per conoscere quali antenne sono disponibili è fornita la funzione *colibriDongleGetDeviceList()*. Dopo aver creato un'handle relativa ad un Colibrì mediante la funzione *colibriCreate()* è possibile rilevare i sensori disponibili mediante la funzione *colibriDongleGetActiveSensorList()*. La funzione *colibriOpenWireless()* apre la connessione tra la dongle-handle

e l'handle del sensore. A questo punto è possibile accedere al Colibrì ed utilizzare tutte le funzioni fornite nelle API. Per avviare e fermare le misurazioni sono necessarie le funzioni *colibriStart()* e *colibriStop()*, che vengono usate parallelamente alle corrispondenti *colibriDongleStart()* e *colibriDongleStop()* relative all'antenna. La connessione al sensore viene chiusa mediante la funzione *colibriClose()* e quella con l'antenna mediante *colibriDongleClose()*. A questo punto è possibile eliminare l'handle mediante *colibriDestroy()* e la dongle-handle mediante *colibriDongleDestroy()*. Segue un esempio di codice C++ utilizzato per stabilire una connessione wireless:

```
void * imu;
void * dongle;
TrivisioDongleSensorEntry wlessID[10];
TrivisioSensor dongleList;
if (!colibriDongleGetDeviceList(&dongleList, 1)) return false;
dongle = colibriDongleCreate();
if (colibriDongleOpen(dongle, 0, dongleList.dev)) return false;
// connessione con antenna stabilita
if (!(sensors = colibriDongleGetActiveSensorList(dongle,
                                                 wlessID, 10)))
          return false;
imu = colibriCreate(0);
if (colibriOpenWireless(imu, dongle, 0, wlessID[0].pos))
   return false;
// connessione con sensore stabilita
colibriDongleStart(dongle);
// trasferimento dati antenna avviato
colibriStart(imu);
// misurazioni avviate, possibile accedere ai dati
colibriStop(imu);
// misurazioni stoppate
colibriDongleStop(dongle);
//trasferimento dati antenna fermato
colibriClose(imu);
// connessione chiusa
colibriDongleClose(dongle);
//connessione antenna chiusa
colibriDongleDestroy(dongle);
colibriDestroy(imu);
//variabili eliminate
```

Capitolo 3

External per Pure Data ht_colibri

3.1 Pure Data

Poiché il contesto in cui verrà utilizzato il sensore riguarda principalmente l'elaborazione di segnali audio, l'ambiente più indicato risulta essere Pure Data. Abbreviato spesso in *PD*, è un progetto open-source realizzato negli anni 90 da Miller Puckette in continua evoluzione e rappresenta una valida alternativa all'analogo programma MAX realizzato dallo stesso professore statunitense per la IRCAM. In *PD* la programmazione è di tipo grafico real-time e consiste nella realizzazione di cosiddette "patch" nella quale vengono collocate le "entità" e collegate tra di loro. Le entità possono essere di quattro tipi: oggetti, messaggi, GUI (tra cui simboli e numeri) e commenti. Gli oggetti possono interagire tra di loro mediante gli ingressi e le uscite chiamati rispettivamente inlets ed outlets. I messaggi possono contenere stringhe di caratteri, numeri, liste o variabili e vengono attivati con un click del mouse oppure alla ricezione di un altro messaggio o di particolari azioni. Prima di passare oltre è utile approfondire in cosa consiste un "messaggio" in *PD*. Esso è costituito da un "selector" e da una lista di "atoms". Gli atom possono essere di tre tipi: A_FLOAT (che indica un valore numerico), A_SYMBOL (che indica una stringa) e A_POINTER (che indica un puntatore). Un selector è un simbolo che definisce il tipo del messaggio e può essere:

- "bang", un "bang"-message non contiene atoms ed indica un bang in *PD*
- "float", un "float"-message contiene un solo atom di tipo A_FLOAT
- "symbol", un "symbol"-message contiene un solo atom di tipo A_SYMBOL
- "pointer", un "pointer"-message contiene un solo atom di tipo A_POINTER
- "list", un "list"-message contiene una lista di uno o più atom di tipo arbitrario.

Gli oggetti sono gli elementi più importanti in una patch, in quanto permettono di manipolare dati o segnali ricevuti in ingresso e restituire un insieme anche eterogeneo di uscite. Per convenzione gli oggetti in grado di gestire segnali presentano nel loro nome il simbolo tilde prima dell'estensione ed i collegamenti che "trasportano" i segnali sono visivamente più spessi.

Un messaggio molto importante ed usato è il cosiddetto "bang" (una sorta di impulso), che consente di "innescare" un'azione; le number box (numeri) invece inviano dei messaggi di tipo numerico. Gli oggetti possono infine comunicare dati in uscita mediante gli outlets, segnali audio attraverso l'uscita audio del computer oppure sfruttare la standarderror di *PD*. Un esempio di patch è visibile in Figura 3.1.

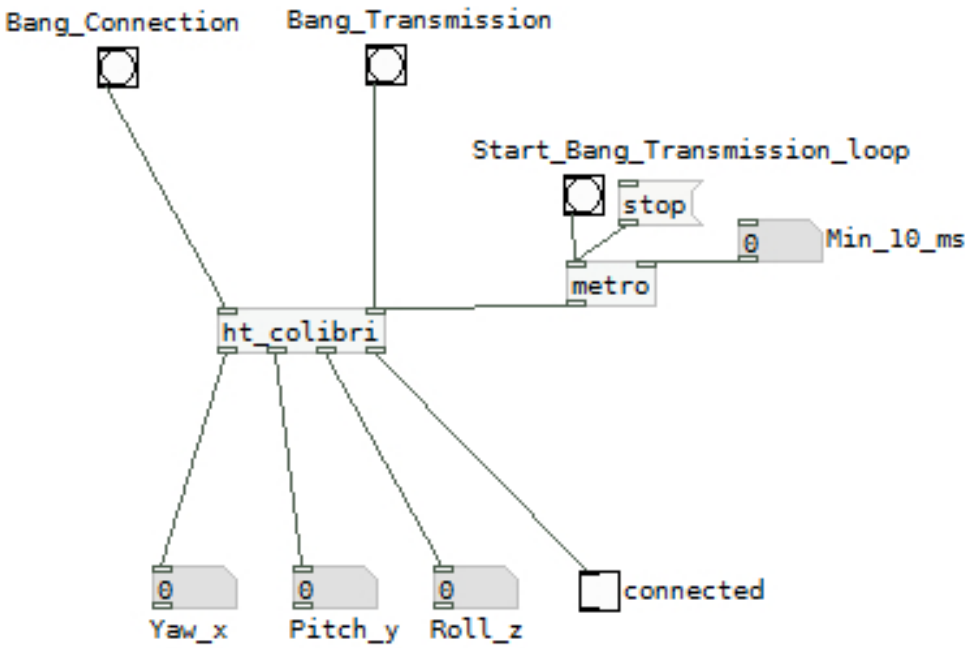

Figura 3.1: *Esempio di Patch PD*

3.2 External per *PD* in Eclipse

Nonostante la grande quantità di oggetti già disponibili in *PD*, come in questo caso si rende necessaria la possibilità di crearne di nuovi. Pure Data prevede che le "external" (le classi non fornite con *PD* ma create da terzi, contrapposte alle "internals" fornite invece nel pacchetto *PD*) vengano scritte in linguaggio C. Tuttavia, per venire incontro alle esigenze del programmatore e semplificare la scrittura del codice, qualche anno fa è stato introdotto Flext, una libreria che consente la scrittura di codice in C++ che si integra con *PD*. Purtroppo lo sviluppo di Flext si è interrotto parecchio tempo fa e di conseguenza la comunità di Pure Data ha ufficialmente deprecato il supporto a tale progetto. La soluzione adottata per la realizzazione del lavoro di tesi è stata perciò quella di ricorrere ad un ambiente di programmazione C++ per mezzo del noto programma Eclipse, sfruttando un apposito template, in grado di gestire l'interazione tra *PD* ed il linguaggio in questione. Per poter procedere in questo modo è necessario modificare adeguatamente il makefile fornito nel template specificando nei punti indicati dagli sviluppatori:

- le directory di installazione di pd
- il nome del file sorgente C++
- il nome dell'oggetto PD prodotto
- ulteriori librerie utilizzate attraverso il comando "-l" (Per il lavoro di tesi svolto ad esempio stata aggunta la stringa "-ltrivisio")

La compilazione del progetto in eclipse produce una dll che descrive il funzionamento dell'oggetto in *PD* che la richiama.

Per scrivere una external in C/C++ occorre servirsi di un'interfaccia tra linguaggio C e Pure Data; la libreria m_pd.h fornisce le funzionalità richieste. Nel seguito verranno descritte le componenti necessarie per un corretto sviluppo:

- Dichiarazione della nuova classe:

  esempio:
  ```
  static t_class * myclass_class;
  ```
  Nell'esempio myclass_class sarà il puntatore alla nuova classe.

- Inizializzazione dataspace:

  esempio:
  ```
  typedef struct _myclass {
  t_object x_ob;
  } t_myclass;
  ```
  La struttura `t_myclass` del tipo `_myclass` costituisce il dataspace della calsse:
  La variabile `x_ob` del tipo `t_object` è assolutamente necessaria, immagazzina le proprietà interne dell'oggetto come la presentazione grafica in *PD*, deve essere la prima definita e può essere seguita da altre.

- Dichiarazione del metodo di setup (il primo che *PD* richiama al posizionamento dell'oggetto) il quale crea la classe ed indica il metodo costruttore (chiamato immediatamente), quello distruttore (chiamato quando l'oggetto viene cancellato in *PD*) e lo spazio necessario all'istanziamento della classe. Contiene inoltre l'elenco di tutti i metodi richiamati alla ricezione di ogni determinato *messaggio* in ingresso. Questo un esempio di metodo di setup:

```
void myclass_setup(void) {
myclass_class = class_new(gensym(``myclass``),
(t_newmethod)myclass_new,
0, sizeof(t_myclass),
CLASS_DEFAULT, (t_atomtype) 0);
class_addbang(myclass_class, myclass_bang);
}
```

  Il metodo `class_new` crea una nuova classe: il primo argomento indica il nome simbolico della classe, il secondo indica il metodo costruttore (`myclass_new` nell'esempio), il terzo il distruttore (non presente nell'esempio), il quarto le dimensioni della data structures (nell'esempio quelle della struttura definita sopra), il quinto le caratteristiche grafiche dell'oggetto in *PD* (nell'esempio quelle di default) e a seguire una lista di massimo sei elementi che definiscono gli argomenti della classe e il loro tipo (A_DEFFLOAT per argomento numerico, A_DEFSYMBOL per argomento simbolico e A_GIMME per una lista di più di sei argomenti) conclusa con uno zero.
  Il metodo `class_addbang` indica che alla ricezione di un messaggio di tipo bang verrà chiamato il metodo `myclass_bang` definito in seguito. Di fatto aggiunge il metodo `myclass_bang` alla classe `myclass_class`. Altri metodi usati per "collegare" messaggi ricevuti e metodi richiamati verranno descritti nei paragrafi successivi.

- Definizione di tutti i metodi nominati precedentemente ed eventualmente altri.
  Il metodo costruttore deve essere di tipo `void *`, principalmente inizializza la variabile `x` di tipo `t_myclass *` e definisce eventuali nuovi inlet ed outlet.
  Relativamente agli esempi fatti finora, in questa parte vanno definiti `myclass_new` e `myclass_bang`:

```
void * myclass_new(void) {
t_myclass * x = (t_myclass *)pd_new(myclass_class);
return (void *)x;
}

void myclass_bang(t_myclass*x) {
post(``Hello world !!``);
}
```

Si segnalano infine le funzioni `post` ed `error`, analoghe alla funzione *printf* di C (della quale ereditano anche gli specificatori), che stampano messaggi sulla *standarderror* di *PD*. `error` in particolare segnala il messaggio stampato come errore.

La descrizione della programmazione in eclipse di external per *PD* non verrà approfondita, sono state comunque fornite le informazioni necessarie alla comprensione del lavoro di tesi svolto. Per una descrizione più approfondita si faccia riferimento alla guida “How to Write an External for Pure Data“ di Johannes M Zmölnig.

3.3 ht_colibri

3.3.1 Descrizione

L'obiettivo principale dell'oggetto ht_colibri posizionato in *PD* è fornire i tre angoli di eulero relativi a Yaw, Pitch e Roll del sensore Colibrì Wireless collocato sulle cuffie. A questo scopo l'external prevede:

- Un inlet che riceve
  - I messaggi di tipo symbol per la gestione delle impostazioni del sensore e in generale dei dati relativi alla connessione.
  - I messaggi di tipo bang (in seguito chiamati *Bang Connection*) per l'apertura e la chiusura della connessione tra il computer ed il sensore
- Un inlet che riceve
  - I messaggi di tipo bang (in seguito chiamati *Bang Transmission*) per la comunicazione dei valori dei tre angoli di eulero relativi all'ultima misurazione effettuata dal sensore. A questo inlet verrà collegata l'uscita di un oggetto metro, in grado di generare segnali di bang a frequenza definibile. Poiché la frequenza di campionamento interna del sensore è di 100 Hz è inutile scendere oltre ai 10ms di periodo.
- Tre outlets che comunicano:
  - Il valore di Yaw
  - Il valore di Pitch
  - Il valore di Roll
- Un outlet che segnala lo stato della connessione mediante un messaggio di tipo float che assume il valore 1 quando è aperta una connessione ad un sensore ed il valore zero altrimenti.

Il file help relativo all'oggetto ht_colibri mostrato in Figura 3.2 descrive come utilizzare l'external nel modo più completo.

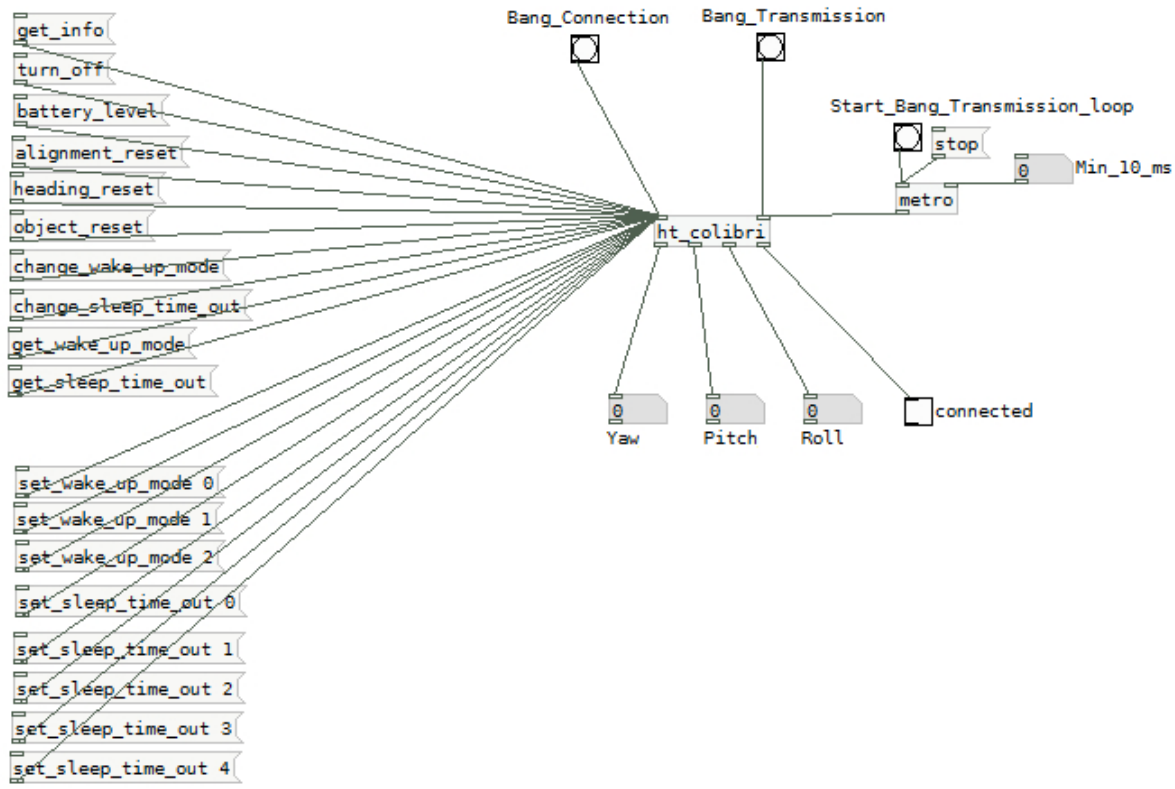

Figura 3.2: *File Help dell'external ht_colibri*

I messaggi in ingresso gestiti sono i seguenti:

- `get_info` – per ottenere informazioni sull'antenna in uso (in caso di connessione wireless) e sul sensore attualmente connesso
- `turn_off` – per spegnere il sensore (solo se connesso via wireless)
- `battery_level` – per ottenere il livello di batteria residua, sia wired che wireless
- `reset_alignment` – per resettare l'allineamento nella posizione attuale. Vengono azzerati Yaw, Pitch e Roll.
- `heading_reset` – per resettare solamente il dato di Yaw lasciando allineati nel modo precedente i dati di Pitch e Roll
- `object_reset` – per resettare i dati di Pitch e Roll lasciando allineato nel modo precedente il dato di Yaw .

- `get_wake_up_mode` – per conoscere la modalità di risveglio in uso
- `change_wake_up_mode` – per passare alla modalità di risveglio successiva
- `set_wake_up_mode f` – per selezionare la modalità di risveglio "f" con f intero da 0 a 2
- `get_sleep_time_out` – per conoscere la modalità di sleep time out in uso
- `change_sleep_time_out` – per passare alla modalità di sleep time out successiva
- `set_sleep_time_out f` – per selezionare la modalità di sleep time out "f" con f intero da 0 a 4

Tutti i metodi restituiscono un messaggio di errore nella *standarderror* se vengono inviati quando non è connesso alcun sensore al computer. Ad eccezione dei metodi relativi ai vari tipi di riallineamento, tutti gli altri cominciano fermando le misurazioni all'interno del sensore e finiscono riavviandole, in modo da poter accedere ai dati necessari evitando che il sensore si blocchi.

`get_info`
Sfrutta i metodi *colibriGetCongleConfig* e *colibriGetConfig* e comunica, in caso di connessione wireless, i seguenti dati relativi all'antenna: ID, freq, ASCII output, autoStart, count, radio channel; ed i seguenti, per qualsiasi tipo di connessione, relativi al sensore: ID, freq, ASCII output, autoStart, raw mode e Jitter reduction.

`turn_off`
Sfrutta i metodi *colibriSetWirelessWakeUpMode* e *colibriSetWirelessSleepTimeOut* e resituisce un messaggio di errore nella standarderror di *PD* nel caso in cui la connessione al sensore non fosse di tipo wireless. Di fatto imposta la modalità di risveglio 0, ovvero solo mediante connessione con il cavo USB, la modalità di sleep time out 0, quindi sleep mode immediato, e chiude la connessione con il sensore che risulta quindi spento.

`battery_level`
Sfrutta i metodi *colibriGetWirelessBatteryLevel* e *colibriCalcWirelessChargeRatio* e comunica in percentuale la batteria residua mediante la *standarderror* di *PD*.

`reset_alignment`, `object_reset` e `heading_reset`
Sfruttano il metodo *colibriBoresight* passando come parametro rispettivamente COLIBRI_ALIGNMENT_RESET, COLIBRI_HEADING_RESET e COLIBRI_HEADING_RESET.

`get_wake_up_mode`, `change_wake_up_mode`, `set_wake_up_mode f`, `get_sleep_time_out`, `change_sleep_time_out`, `set_sleep_time_out f`
Sfruttano a dovere i metodi *colibriGetWirelessWakeUpMode*, *colibriSetWirelessWakeUpMode*, *colibriGetWirelessSleepTimeOut* e *colibriSetWirelessSleepTimeOut*.

`Bang Connection`

Alla ricezione di un bang sul primo inlet viene richiamata la funzione *init()* che cerca di stabilire inizialmente una connessione cablata ed in secondo luogo una connessione wireless. In caso di rilevamento di un sensore la funzione apre la connessione, imposta il sensore nel modo più indicato e termina avviando le misurazioni nel sensore mediante il metodo *colibriStart*. A questo punto il led del sensore comincia a lampeggiare indicando che è possibile ricevere dati relativi alle misurazioni.

`Bang Transmission`

Alla ricezione di un bang nel secondo inlet, se non vi sono sensori connessi viene visualizzato un messaggio di errore nella *standarderror* di *PD*, altrimenti vengono richiesti al sensore i dati relativi alle ultime misurazioni eseguite attraverso il metodo *colibriGetData* e processati mediante la funzione *colibriEulerOri*.

3.3.2 Codice

Il contenuto del file sorgente C++ dell'external ht_colibri progettata è presente in Appendice A. Si ritiene che la descrizione dei metodi utilizzabili già presentata unita ai commenti presenti nel codice sia sufficiente per la comprensione dello stesso, ad ogni modo vengono indicate di seguito le principali decisioni effettuate:

- la variabile booleana `connected` assume il valore *true* solo quando una connessione stata correttamente stabilita. Di fatto l'outlet "connected" varia assieme ad essa.
- la variabile booleana `wless` viene inizializzata al valore *false* e diventa *true* solo dopo il fallimento di un tentativo di connessione cablato. A questo punto viene tentata la connessione ad un sensore wireless ed in caso di successo la variabile in questione rimane *true* (indicando quindi la natura della connessione), altrimenti ritorna false e nessuna connessione stata connessa.
- la variabile booleana `frei` assume il valore *false* solo quando sono in corso modifiche alle impostazioni del sensore. In questo modo l'accesso ai dati misurati viene momentaneamente bloccato per evitare di accedere ad essi quando le misurazioni non sono effettivamente in corso.
- quando una connessione viene stabilita vengono impostate le modalità Wake Up Mode 1e Sleep Time Out 0, le quali favoriscono il miglior risparmio della batteria ed un facile risveglio del sensore (tramite shake delicato).
- se vengono rilevati più sensori in fase di connessione ciò viene segnalato e viene utilizzato il primo sensore della colibrì network.
- quando la connessione viene chiusa le correnti impostazioni vengono salvate nel sensore.
- quando l'oggetto ht_colibri viene cancellato da *PD* mentre una connessione è attiva, quest'ultima si chiude correttamente in modo automatico dopo aver fermato le misurazioni.

Capitolo 4

Conclusioni

4.1 Conclusioni

Il lavoro svolto ha raggiunto in maniera soddisfacente gli obiettivi prefissati. L'external di *PD* funziona bene e i dati forniti in uscita coincidono, per quanto riguarda gli errori sul rilevamento degli angoli di orientazione, con quelli dichiarati dal costruttore. Poiché il lavoro svolto nel progetto del gruppo SMC dell'Università prevedeva già un'implementazione di questo tipo, è stato possibile testare il sensore su una patch di prova in grado di elaborare un segnale audio tenendo conto dell'head tracking realizzato. Nell'utilizzo dell'external progettata bisogna tenere conto dell'orientazione con cui il sensore viene fissato alla testa dell'utente ed in caso elaborare i tre dati in uscita in modo adeguato prima di usufruirne. I risultati sono stati soddisfacenti, la posizione della testa viene correttamente rilevata ed una prima spazializzazione del suono è stata udibile in cuffia.
I campi di utilizzo possono essere tra i più disparati e non è possibile attualmente delinearne tutti gli sviluppi futuri; tuttavia la ricerca si indirizza già verso:

- la realizzazione di conferenze virtuali dotate di spazialità sonora
- l'implementazione di cuffie con head tracker nei video games per un grado di immersione maggiore
- la realizzazione di guide virtuali sonore provenienti dalle varie opere di un museo (applicazione denominata AR - Augmented Reality - realtà aumentata, nel quale mondo reale e mondo virtuale si sovrappongono)
- in generale la creazione dei VAS citati in precedenza che permettono la realizzazione di scene sonore virtuali, come ad esempio la simulazione di un "tour sonoro" in mezzo ai vari musicisti di un orchestra simulata

L'head-tracking ad ogni modo, non prevede un campo di applicazione così mirato: esso può infatti trovare il suo spazio anche al di fuori dell'esperienza audio spazializzato citata fino ad ora, ad esempio può essere utilizzato in ambiente multimodale audio e video, nel quale viene realizzata anche una visione stereoscopica dinamica oltre all'elaborazione sonora descritta.

Appendici

Appendice A

Codice C++ dell'external ht_colibri

A.1 ht_colibri.cpp

```
/* code for "ht_colibri" pd class.  This manages the connection
 between the computer and a sensor Colibri Wireless. */
#include <m_pd.h>
#include "TrivisioColibri.h"
#include "TrivisioWirelessColibri.h"

#define M_PI 3.14159265358979323846
// wless=true indicates a wireless connection;
// frei=false when colibri settings are being modifyied:
// connected=true if a sensor is connected
bool wless=false,connected=false, frei=true;

// imu: handle to associate with the sensor
void* imu = 0;

// dongle: dongle-handle to associate with the dongle
void* dongle = 0;
// dList and dconf are used to export the variables
// "dongleList" and "conf" created into the funcion init().
TrivisioSensor dList;
DongleConfig dconf;

TrivisioSensor sensorList[10]; // used to discover wired sensors
TrivisioIMUData data; // used to contain measurements data of the sensor
ColibriConfig conf; // used to configure the sensor
int sensors=0;// found sensors counter
TrivisioDongleSensorEntry wlessID[10];// used to discover wireless sensors


/* this is a pointer to the class for "ht_colibri", which is created in the
 "setup" routine below and used to create new ones in the "new" routine. */
t_class *ht_colibri_class;
```

```
/* the data structure for each copy of "ht_colibri".
 Specifies variables of pd types Like inlet and outlet.
 3 outlet for Yaw, Pitch, Roll and 1 outlet for connection flag */
typedef struct ht_colibri
{
    t_object x_ob;
    t_outlet *x_out, *y_out, *z_out, *connected_out;
} t_ht_colibri;

/* when ht_colibri is deleted from pd, it correctly
 closes the connection with the sensor if opened. */
static void ht_colibri_delete(void)
{
    if (connected)
        {
        if (wless)
            colibriDongleStop(dongle);  //  stop dongle measurements
        colibriStop(imu);               //  stop colibri measurements
        colibriClose(imu);              //  close colibri connection
        colibriDestroy(imu);            //  destroy variable
        if (wless)
            colibriDongleClose(dongle); // close dongle connection
        colibriDongleDestroy(dongle);   // destroy variable
        connected=false;
        wless=false;
        post("Colibri: Disconnected");
        post("----------------------");
        post("----------------------");
        }
    post("ht_colibri_delete");
}

/* every message method starts stopping measurements to access
 colibri settings and finishes restarting them
 When a "battery_level" message is received the battery level
 is printed it in the standarderror */
void ht_colibri_battery_level(t_ht_colibri *x)
{
    post("Colibri: Battery_Level");
    if (connected)
        {
        frei=false;
        float l;
        if (wless) colibriDongleStop(dongle);
        colibriStop(imu);
l= (colibriCalcWirelessChargeRatio(colibriGetWirelessBatteryLevel(imu)))*100;
        post("Battery Level = %2.2f %s",l, " % ");
        if (wless)
            colibriDongleStart(dongle);
        colibriStart(imu);
        frei=true;
```

```
        }
    else error("No Sensor Connected");
    post("----------------------");
}

/* When a "alignment_reset" message is received
 the boresight is setted properly
 (boresighting doesn't need to stop measurements) */
void ht_colibri_alignment_reset(t_ht_colibri *x)
{
    post("Colibri: Alignment_Reset");
    if (connected)
        {
        frei=false;
        colibriBoresight(imu, COLIBRI_ALIGNMENT_RESET);
        post("Done");
        frei=true;
        }
    else error("No Sensor Connected");
    post("----------------------");

}

// When a "heading_reset" message is received the boresight is setted properly
void ht_colibri_heading_reset(t_ht_colibri *x)
{
    post("Colibri: Heading_Reset");
    if (connected)
    {
        frei=false;
        colibriBoresight(imu, COLIBRI_HEADING_RESET);
        frei=true;
    }
    else error("No Sensor Connected");
    post("----------------------");
}

// When a "object_reset" message is received the boresight is setted properly
void ht_colibri_object_reset(t_ht_colibri *x)
{
    post("Colibri: Object_Reset");
    if (connected)
    {
        frei=false;
        colibriBoresight(imu, COLIBRI_OBJECT_RESET);
        frei=true;
    }
    else error("No Sensor Connected");
    post("----------------------");
}
```

```
/* When a "change_wake_up_mode" message is received
 the next wake up mode is setted */
void ht_colibri_change_wake_up_mode(t_ht_colibri *x)
{
    post("Colibri: Change_Wake_Up_Mode");
    if (connected)
    {
        frei=false;
        if (wless)
            colibriDongleStop(dongle);
        colibriStop(imu);
        int m,l;
        m = colibriGetWirelessWakeUpMode(imu);
        l = m%2 - m/2 +1;                          // 0-->1 1-->2 2-->0
        colibriSetWirelessWakeUpMode(imu,l);
        if (l==0) post ("Mode 0: Only by USB");
        if (l==1) post ("Mode 1: By USB or tapping the sensor slightly");
        if (l==2) post ("Mode 2: From Wireless Communication,
                                    Power consumption reduced to 1/3");
        if (wless)
            colibriDongleStart(dongle);
        colibriStart(imu);
        frei=true;
    }
    else error("No Sensor Connected");
    post("----------------------");
}

// When a "set_wake_up_mode" message is reeived the right wake up mode is setted
void ht_colibri_set_wake_up_mode(t_ht_colibri *x,t_floatarg f)
{
    post("Colibri: Set_Wake_Up_Mode");
    if (connected)
    {
        if ((f>=0)&(f<=2)&(f==f/1))
        {
            frei=false;
            if (wless)
                colibriDongleStop(dongle);
            colibriStop(imu);
            colibriSetWirelessWakeUpMode(imu,f);
            if (f==0) post ("Mode 0: Only by USB");
            if (f==1) post ("Mode 1: By USB or tapping the sensor slightly");
            if (f==2) post ("Mode 2: From Wireless Communication,
                                                Power consumption reduced to 1/3");
            if (wless)
                colibriDongleStart(dongle);
            colibriStart(imu);
            frei=true;
        }
        else error ("Invalid argument, only integers from 0 to 4 are admitted");
```

```
    }
    else error("No Sensor Connected");
    post("----------------------");
}

/* When a "change_sleep_time_out" message is received
 the next sleep time out mode is setted */
void ht_colibri_change_sleep_time_out(t_ht_colibri *x)
{
    post("Colibri: Change_Sleep_Time_Out");
    if (connected)
    {
        frei=false;
        if (wless) colibriDongleStop(dongle);
        colibriStop(imu);
        int m,l;
        m= colibriGetWirelessSleepTimeOut(imu);
        l = m%4 - m/4 +1;
        colibriSetWirelessSleepTimeOut(imu,l);
        if (l==0) post ("Mode 0: Immediate Turn Off");
        if (l==1) post ("Mode 1: 1 Min");
        if (l==2) post ("Mode 2: 2 Min");
        if (l==3) post ("Mode 3: 5 Min");
        if (l==4) post ("Mode 4: 10 Min");
        if (wless)
            colibriDongleStart(dongle);
        colibriStart(imu);
        frei=true;
    }
    else error("No Sensor Connected");
    post("----------------------");
}

/* When a "set_sleep_time_out" message is received
 the right sleep time out mode is setted */
void ht_colibri_set_sleep_time_out(t_ht_colibri *x,t_floatarg f)
{
    post("Colibri: Set_Sleep_Time_Out");
    if (connected)
    {
        if ((f>=0)&(f<=4)&(f==f/1))
        {
            frei=false;
            if (wless) colibriDongleStop(dongle);
            colibriStop(imu);
            colibriSetWirelessSleepTimeOut(imu,f);
            if (f==0) post ("Mode 0: Immediate Turn Off");
            if (f==1) post ("Mode 1: 1 Min");
            if (f==2) post ("Mode 2: 2 Min");
            if (f==3) post ("Mode 3: 5 Min");
            if (f==4) post ("Mode 4: 10 Min");
```

```
            if (wless)
                colibriDongleStart(dongle);
            colibriStart(imu);
            frei=true;
        }
        else error ("Invalid argument, only integers from 0 to 2 are admitted");
    }
    else error("No Sensor Connected");
    post("----------------------");
}

/* When a "get_wake_up_mode" message is received
 the current wake up mode is printed in the standarderror */
void ht_colibri_get_wake_up_mode(t_ht_colibri *x)
{
    post("Colibri: Get_Wake_Up_Mode");
    if (connected)
    {
        frei=false;
        if (wless) colibriDongleStop(dongle);
        colibriStop(imu);
        int m;
        m = colibriGetWirelessWakeUpMode(imu);
        if (m==0) post ("Mode 0: Only by USB");
        if (m==1) post ("Mode 1: By USB or tapping the sensor slightly");
        if (m==2) post ("Mode 2: From Wireless Communication,
                                            Power consumption reduced to 1/3");
        if (wless)
            colibriDongleStart(dongle);
        colibriStart(imu);
        frei=true;
    }
    else error("No Sensor Connected");
    post("----------------------");
}

/* When a "get_sleep_time_out" message is received
 the current sleep time out mode is printed in the standarderror */
void ht_colibri_get_sleep_time_out(t_ht_colibri *x)
{
    post ("Colibri: Get_Sleep_Time_Out");
    if (connected)
    {
        frei=false;
        if (wless) colibriDongleStop(dongle);
        colibriStop(imu);
        int m;
        m = colibriGetWirelessSleepTimeOut(imu);
        if (m==0) post ("Mode 0: Immediate Turn Off");
        if (m==1) post ("Mode 1: 1 Min");
        if (m==2) post ("Mode 2: 2 Min");
```

```
        if (m==3) post ("Mode 2: 5 Min");
        if (m==4) post ("Mode 2: 10 Min");
        if (wless)
            colibriDongleStart(dongle);
        colibriStart(imu);
        frei=true;
    }
    else error("No Sensor Connected");
    post("----------------------");
}

/* When a "turn_off" message is received,
 if a wireless sensor is connected,
 it stops the measurements, sets the wake up mode to 0,
 sets sleep time out to 0 and closes the connection */
void ht_colibri_turn_off(t_ht_colibri *x)
{
    post ("Colibri: Turn_Off");
        if (connected)
    {
        if (wless)
                {
            frei=false;
            colibriDongleStop(dongle);
            colibriStop(imu);
// save settings in colibri before setting wake up mode and sleep time out
            if(colibriSaveSettings(imu)==0)
                post ("Settings saved");
            else
                post("Error while saving settings");
            colibriSetWirelessSleepTimeOut(imu,0);
            colibriSetWirelessWakeUpMode(imu,0);
            colibriClose(imu);
            colibriDestroy(imu);
            if (wless)
                colibriDongleClose(dongle);
            colibriDongleDestroy(dongle);
            connected=false;
            wless=false;
            post("Sensor disconnected and turned off, to turn on
                                                connect the sensor by USB");
            outlet_float(x->connected_out, 0);

                }
        else error ("Use turn_off only with wireless connection");
    }
    else error ("No Sensor Connected");
    post("----------------------");
}

/* When a "get_info" message is received, some main info about
```

```
 dongle and sensor are printed in the standarderror */
void ht_colibri_get_info(t_ht_colibri *x)
{
    post ("Colibri: Get_Info");
    if (connected)
    {
        frei=false;
        if (wless)
            colibriDongleStop(dongle);
        colibriStop(imu);
        if (wless)
        {
            post("Colibr Wireless Dongle ----------------");
            post("Device ID:        %s",dList.ID);
            post("Frequency:        %i",dconf.freq);
            post("ASCII output:     %s",(dconf.ascii)?"true":"false");
            post("autoStart:        %s",(dconf.autoStart)?"true":"false");
            post("count:            %s",(dconf.count)?"true":"false");
            post("radio channel:    %i",int(dconf.radioChannel));
            post("---------------------------------------");
        }
        post("Colibr IMU ");
        if (wless)
            post("Device ID:        %s",wlessID[0].ID);
        else
            post("Device ID:        %s",sensorList[0].ID);
        post("Sensor config:    0x %X",conf.sensor);
        post("Frequency:        %i",conf.freq);
        post("ASCII output:     %s",(conf.ascii)?"true":"false");
        post("autoStart:        %s",(conf.autoStart)?"true":"false");
        post("RAW mode:         %s",(conf.raw)?"true":"false");
        bool t;
        if (colibriGetJitterStatus(imu)!=0) {t=true;} else t=false;
        post("Jitter reduction: %s",(t)?"Enabled":"Disabled");
        post("---------------------------------------");
        if (wless)
            colibriDongleStart(dongle);
        colibriStart(imu);
        frei=true;
    }
    else error("No Sensor Connected");
    post("--------------------");
}

/* init Makes an attempt to connect to a wired sensor.
 If it fails, it makes an attempt to connect to a wireless sensor.
 Returns true on success. */
bool init()
{
    static float Ka[9] = { // Diagonal matrices with diagonal element
        // .68 yelds approx 20Hz // bandwidth @ 100 Hz
```

```
    0.68f,    0.00f,    0.00f,
    0.00f,    0.68f,    0.00f,
    0.00f,    0.00f,    0.68f
};
static float Kg[9] = {
    0.68f,    0.00f,    0.00f,
    0.00f,    0.68f,    0.00f,
    0.00f,    0.00f,    0.68f
};
post("Trying to connect to a wired sensor");
if (!(sensors = colibriGetDeviceList(sensorList, 10)))
    // to discover wired sensors connected to the computer
{
    post("No wired sensors found");
    post("Trying to connect to a wireless sensor");
    wless=true;
}
if ((sensors<0)||(sensors>1))
{
    post("More than a wired sensor were found, the first one will be used");
}
if (wless)
{
    TrivisioSensor dongleList;
    if (!colibriDongleGetDeviceList(&dongleList, 1))
        // looking for a dongle connected
    {
        post ("Colibri Dongle not found");
        return false;
    }
    post ("Colibri Dongle found");
    dongle = colibriDongleCreate();
    // dongle-handle, represents the dongle
    if (colibriDongleOpen(dongle, 0, dongleList.dev))
    {
        error("Error while trying to access Colibri Dongle: %s",
                                                        dongleList.ID);
        return false;
    }
    DongleConfig conf;
    colibriDongleGetConfig(dongle, &conf);
    conf.freq = 100;
    conf.ascii = false;
    conf.count = false;
    colibriDongleSetConfig(dongle, &conf);
    dList = dongleList;
    dconf = conf;
    if (!(sensors = colibriDongleGetActiveSensorList(dongle, wlessID, 10)))
        // to discover wireless sensors active in the dongle radio channel
    {
        error( "No wireless sensors found");
```

```
        return false;
    }
    if ((sensors<0)||(sensors>1))
    {
    post("More than a wireless sensor found, the first one will be used");
    }
}
try
{
    imu = colibriCreate(0);//  colibri handle, represents the sensor
    if (wless)
    {
        if (colibriOpenWireless(imu, dongle, 0, wlessID[0].pos))
            // open connection between dongle and sensor
        {
        error("Error while trying to access Wireless Colibri: %s",
                                                          wlessID[0].ID);
        colibriDongleDestroy(dongle);
        return false;
        }
    }
    else
    {
    if (colibriOpen(imu, 0, sensorList[0].dev))
        error("Error while trying to access Wired Colibri: %s",
                                                      sensorList[0].ID);
        // open connection with the wired sensor
    }
    bool wlessHW = (wless||(sensorList[0].type==TrivisioSensor::COLIBRI_W));
    colibriGetConfig(imu, &conf);
    colibriSetJitterStatus(imu, true);
    colibriSetWirelessWakeUpMode(imu,1);
    colibriSetWirelessSleepTimeOut(imu,0);
    conf.raw = false;
    conf.magDiv = wlessHW?1:3;
    conf.freq = 100;
    conf.ascii = false;
    conf.sensor = ColibriConfig::ALL;
    colibriSetConfig(imu,&conf);
    colibriSetKa(imu, Ka);
    colibriSetKaStatus(imu, false);
    colibriSetKg(imu, Kg);
    colibriSetKgStatus(imu, false);
}
catch(...)
{
error("Failed to initialize sensor:  %s",(wless)?
                                    (wlessID[0].ID):(sensorList[0].ID));
}
if (wless)
    colibriDongleStart(dongle);    // start dongle measurements
```

```
    colibriStart(imu);                  // start colibri measurements
    post("Connected to sensor: %s", (wless)?(wlessID[0].ID):(sensorList[0].ID));
    return sensors!=0;
}

/* When a bang message is received at the first inlet,
 if a sensor is connected it will be disconnect and vice versa */
void ht_colibri_bangconnection(t_ht_colibri *x)
{
    post ("Colibri: Bang Connection");
    if (!connected) // if disconnected --> try to connect
        {
        wless=false;
        if(init())
                {
            connected=true;
            outlet_float(x->connected_out, 1);
                }
        else
            {
            post("Colibri: Attempt Failed");
            post("----------------------");
            }
        }
    else    // if connected -->  disconnect
        {

        if (wless)
            colibriDongleStop(dongle);
        colibriStop(imu);
        if(colibriSaveSettings(imu)==0) // save settings in colibri
            post ("Settings saved");
        else
            post("Error while saving settings");
        colibriClose(imu);
        colibriDestroy(imu);
        if (wless)
            colibriDongleClose(dongle);
        colibriDongleDestroy(dongle);
        connected=false;
        outlet_float(x->connected_out, 0);
        wless=false;
        post("Colibri: Disconnected");
        }
        post("----------------------");
        post("");
}

/* When a bang message is received at the second inlet,
 the 3 oulets are refreshed according to the last
 measurements made by the sensors */
```

```
void ht_colibri_transmissionbang(t_ht_colibri *x)
{
    if (connected)
    {
        if (frei)
        {
            colibriGetData(imu, &data);
            float eul[3],a[3];
            colibriEulerOri(&data, eul);
            a[0]=180/M_PI*eul[0];
            a[1]=180/M_PI*eul[1];
            a[2]=180/M_PI*eul[2];
            outlet_float(x->x_out, a[0]);
            outlet_float(x->y_out, a[1]);
            outlet_float(x->z_out, a[2]);
        }
    }
    else error("No Colibri Sensor Connected");
}

// The constructor initialize the object with inlets and outlets
void *ht_colibri_new(void)
{
    // create pd object class
t_ht_colibri *x = (t_ht_colibri *)pd_new(ht_colibri_class);
connected=false;

// to create the second inlet and call "transmissionbang"
    // if a bang message is received
inlet_new(&x->x_ob,&x->x_ob.ob_pd,gensym("bang"),
                           gensym("transmissionbang"));


// create 3 new outlets for yaw, pitch and roll data
x->x_out = outlet_new(&x->x_ob,&s_float);
x->y_out = outlet_new(&x->x_ob,&s_float);
x->z_out = outlet_new(&x->x_ob,&s_float);

// create new float outlet for the connection flag
x->connected_out = outlet_new(&x->x_ob,&s_float);
post("ht_colibri_new");
return (void *)x;
}
// this is called once at setup time, when this code is loaded into Pd.
extern "C"        {
#ifdef WIN32
    __declspec(dllexport) void ht_colibri_setup(void)
#else
    void ht_colibri_setup(void)
#endif
        {
```

```
post("ht_colibri_setup_headtracker");
ht_colibri_class = class_new(gensym("ht_colibri"), // name of the pd object
                             (t_newmethod)ht_colibri_new, // constructor method
                             ht_colibri_delete, // distructor method
                             sizeof(t_ht_colibri), //size of the class
                             0, // pd object aspect, DEFAULT
                             (t_atomtype) 0); // needed arguments,
                                                 // no one is needed

// each message received is associated with
        //the corresponding called method in the following part
class_addbang(ht_colibri_class, ht_colibri_bangconnection);
class_addmethod(ht_colibri_class, (t_method)ht_colibri_battery_level,
    gensym("battery_level"), (t_atomtype) 0);
class_addmethod(ht_colibri_class, (t_method)ht_colibri_alignment_reset,
    gensym("alignment_reset"), (t_atomtype) 0);
class_addmethod(ht_colibri_class, (t_method)ht_colibri_heading_reset,
    gensym("heading_reset"), (t_atomtype) 0);
class_addmethod(ht_colibri_class, (t_method)ht_colibri_object_reset,
    gensym("object_reset"), (t_atomtype) 0);
class_addmethod(ht_colibri_class, (t_method)ht_colibri_change_wake_up_mode,
    gensym("change_wake_up_mode"), (t_atomtype) 0);
class_addmethod(ht_colibri_class, (t_method)ht_colibri_change_sleep_time_out,
    gensym("change_sleep_time_out"), (t_atomtype) 0);
class_addmethod(ht_colibri_class, (t_method)ht_colibri_set_wake_up_mode,
    gensym("set_wake_up_mode"),A_DEFFLOAT, (t_atomtype) 0);
class_addmethod(ht_colibri_class, (t_method)ht_colibri_set_sleep_time_out,
    gensym("set_sleep_time_out"),A_DEFFLOAT, (t_atomtype) 0);
class_addmethod(ht_colibri_class, (t_method)ht_colibri_get_wake_up_mode,
    gensym("get_wake_up_mode"), (t_atomtype) 0);
class_addmethod(ht_colibri_class, (t_method)ht_colibri_get_sleep_time_out,
    gensym("get_sleep_time_out"), (t_atomtype) 0);
class_addmethod(ht_colibri_class, (t_method)ht_colibri_transmissionbang,
    gensym("transmissionbang"), (t_atomtype) 0);
class_addmethod(ht_colibri_class, (t_method)ht_colibri_turn_off,
    gensym("turn_off"), (t_atomtype) 0);
class_addmethod(ht_colibri_class, (t_method)ht_colibri_get_info,
    gensym("get_info"), (t_atomtype) 0);
        }
}
```

Bibliografia

[1] F. Avanzini, *Algorithms for Sound and Music Computing*, *Capitolo 1*, 2009, reperibile all'indirizzo `http://smc.dei.unipd.it/education.html`.

[2] F. Avanzini e G. De Poli, *Algorithms for Sound and Music Computing*, *Capitolo 4*, 2009, reperibile all'indirizzo `http://smc.dei.unipd.it/education.html`.

[3] M. Geronazzo, S. Spagnol e F.Avanzini, *Customized 3D sound for innovative interaction design*, 2011.

[4] D. R. Begault, E. M. Wenzel and M. R. Anderson, *Direct comparison of the impact of head tracking, reverberation, and individualized head-related transfer functions on the spatial perception of a virtual speech source*, 2001.

[5] *User Manual Colibri Software API*, reperibile all'indirizzo `http://www.trivisio.com/index.php/support/user-manuals`

[6] *Trivisio*, `www.Trivisio.com/`

[7] *Eclipse*, `www.eclipse.org/`

[8] *Pure Data*, `http://puredata.info/`.

[9] *Wiki del progetto pdsmc*, `http://smc.dei.unipd.it/redmine/projects/pdsmc/wiki/ExternalC++`.

[10] *Wikipedia*, `http://en.wikipedia.org/wiki/Euler_angles`.

[11] J. M. Zmölnig, *HOWTO write an External for puredata*, reperibile all'indirizzo `http://pdstatic.iem.at/externals-HOWTO/pd-externals-HOWTO.pdf`.