UNIVERSITÀ DI PADOVA **FACOLTÀ DI INGEGNERIA**

TESI DI LAUREA

SIMULAZIONE NUMERICA DEL CONTRIBUTO ACUSTICO DELLA TESTA NELL'ASCOLTO SPAZIALE: BOUNDARY ELEMENT METHOD

Laureando: Paolo Montesel

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

Corso di Laurea Triennale in Ingegneria dell'Informazione

26 Settembre 2013

Anno Accademico 2012/2013

Prefazione

In questo lavoro di tesi vengono simulate numericamente le Head Related Transfer Function (HRTF) della testa di un manichino KEMAR (Knowles Electronic Manikin for Acoustic Research) per test acustici. Gli obiettivi principali riguardano la valutazione della tecnica Boundary Element Method (BEM) nella risoluzione di problemi acustici non banali e la quantificazione delle risorse computazionali richieste. In particolare, viene testato ed utilizzato il software di simulazione acustica *AcouSTO* e ne vengono trattate caratteristiche e configurazione. Vengono inoltre sviluppati degli strumenti per gestire le simulazioni e alcune tecniche di post-processing su modelli 3D acquisiti tramite scanner. Lo studio preliminare effettuato mostra l'efficacia dei metodi proposti e promuove la loro candidatura per simulazione di scenari più complessi ed accurati.

Sommario

Una *Head Related Transfer Function*, spesso abbreviata in *HRTF*, è una funzione che descrive come un suono proveniente da un punto specifico dello spazio giunga all'orecchio di un ascoltatore.

La conoscenza di tali informazioni permette di capire, e quindi modellare, il modo in cui la riflessione e la diffrazione del suono sul nostro corpo fornisce informazioni spaziali al cervello. In particolare, i contributi più importanti all'ascolto spaziale sono dati dalla forma dell'orecchio esterno, da quella della testa e dalle trasformazioni statiche e dinamiche che introducono nel segnale acustico che giunge al timpano.

Il principale metodo per registrare una *HRTF* è quello di effettuare delle registrazioni su soggetti umani, o manichini, in una camera anecoica. Questo procedimento prevede l'inserimento di un microfono nel canale uditivo del soggetto, che deve restare immobile, e l'utilizzo di una sorgente sonora mobile. La sorgente viene poi spostata per intervalli a diverse distanze ed angolazioni rispetto al soggetto, in modo da valutare l'influenza di tali parametri nella risposta in frequenza. Tutto ciò si traduce in costi, soprattutto in termini di tempo, molto elevati.

L'avanzamento tecnologico degli ultimi anni ha reso possibile l'utilizzo dei computer per simulare numericamente le *HRTF*, la cui trattazione analitica è impraticabile. Inoltre, una volta acquisita la mesh di una testa, è possibile utilizzarla in diversi scenari senza richiedere un ulteriore coinvolgimento del soggetto.

L'obiettivo del lavoro di tesi è quello di valutare la validità di questa tecnica attraverso la simulazione della *HRTF* di un manichino per test acustici.

Nel **primo capitolo** viene esposta brevemente la formulazione matematica del problema della riflessione e della diffrazione delle onde sonore su un corpo. Viene poi trattato il modello di calcolo ottenuto tramite la discretizzazione della superficie dello stesso.

Nel **secondo capitolo** viene introdotto il setup hardware e software utilizzato, con particolare attenzione alle caratteristiche del programma di simulazione *AcouSTO* e al sistema *IBM P770 Power7*. È poi descritto uno scenario di simulazione semplificato di prova che prevede la misura della pressione acustica sulla superficie di una sfera data una sorgente puntiforme posta a distanza ravvicinata.

Nel **terzo capitolo** vengono esposti gli strumenti sviluppati. Essi hanno lo scopo di estendere le funzionalità del programma di simulazione, aumentare la tolleranza ai guasti hardware, generare il file di descrizione dei microfoni, assemblare i file contenenti le risposte in frequenza degli stessi, effettuare delle misurazioni sulla mesh del manichino *KEMAR* e ricreare il modello

completo della testa a partire da una delle sue due metà simmetriche in modo da poter sfruttare tale simmetria per ridurre la complessità del problema.

Nel **quarto capitolo** è descritto lo scenario di simulazione della *HRTF* di una testa, usando come modello un manichino acquisito tramite scanner 3D. L'attenzione è posta sul post-processing applicato alla mesh acquisita e alle considerazioni sulla trattabilità del problema con diverse risoluzioni della stessa.

Il **quinto capitolo** presenta le considerazioni finali e i possibili sviluppi futuri del lavoro svolto.

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Boundary Element Method

Per eseguire simulazioni di HRTFs esistono numerosi metodi [5], che variano in complessità e in costo computazionale. Il metodo preso in considerazione in questa tesi prende il nome di Boundary Elementh Method (BEM) e considera la testa come una mesh di elementi discreti. Fondamentale è l'ipotesi semplificativa che solo la superfice della testa sia da tenere in considerazione; inoltre tale superficie viene considerata come se avesse un'impedenza acustica infinita, quindi tutte le propagazioni attraverso la testa sono ignorate.

1.1 Problema matematico

Il problema acustico [9] è scritto in termini della funzione potenziale di velocità φ nel dominio di Laplace

$$\nabla^2\widetilde{\varphi}(x) - k^2\widetilde{q}(x) = \widetilde{q}, \qquad x \in \Omega \tag{1.1}$$

dove $\widetilde{q}$ rappresenta la sorgente acustica e $k = s/c_0$ è il numero d'onda complesso, essendo $s = \alpha + j\omega$ la variabile di Laplace e c_0 la velocità del suono nelle condizioni di riferimento. Il problema è completato dalle condizioni al contorno per $x \in \partial\Omega$. L'equazione 1.1 vale sia per il potenziale di velocità che per la perturbazione di pressione. Il significato fisico della soluzione è dato dalle condizioni al contorno. Assumendo $\widetilde{\varphi}$ come potenziale di velocità e ponendo $\widetilde{q} = 0$, la formulazione integrale sulla frontiera per $\widetilde{\varphi}$ si può scrivere come

$$E(y)\widetilde{\varphi}(y) = \oint_S \left(G\frac{\partial\widetilde{\varphi}}{\partial n} - \widetilde{\varphi}\frac{\partial G}{\partial n} \right) dS(x) \tag{1.2}$$

dove la funzione di dominio $E(y)$ è

$$\begin{cases} 1 & if \quad y \in \Omega \\ 1/2 & if \quad y \in \partial\Omega \\ 0 & if \quad y \notin \Omega \end{cases} \tag{1.3}$$

e $S = \partial\Omega$ per i problemi interni e $S = \partial\Omega/S_\infty$ per i problemi esterni. Richiamando l'espressione 1.1, la 1.2 diventa

$$E(y)\widetilde{\varphi}(y) = \oint_S \left(G\frac{\partial\widetilde{\varphi}}{\partial n} - \widetilde{\varphi}\frac{\partial G_0}{\partial n} + s\widetilde{\varphi}G_0\frac{\partial\theta}{\partial n} \right) e^{-s\theta} dS(x) \tag{1.4}$$

L'eq 1.4 è usata come una rappresentazione integrale al contorno per $\widetilde{\varphi}$ in un punto arbitrario nello spazio (i microfoni) come funzione della distribuzione di $\widetilde{\varphi}$ sulla frontiera.

L'equazione 1.4 è risolta per via numerica attraverso il Boundary Element Method. La frontiera del dominio è partizionata in N quadrilateri e tutte le quantità sono considerate costanti all'interno di ogni pannello. L'integrale di superficie nell'equazione 1.4 è approssimato con una somma di N pannelli.

1.2 Principio di reciprocità e inversione del problema acustico

Idealmente, per effettuare le misurazioni desiderate, sarebbe necessario predisporre un microfono in ciascun orecchio del manichino. Dopodiché, una singola sorgente sonora andrebbe spostata nello spazio per osservare le modifiche di ampiezza e fase al variare di distanza, azimut ed elevazione.

C'è però un altro modo per ottenere gli stessi risultati riducendo il costo computazionale: usando un teorema analogo a quello di Maxwell-Betti per l'acustica si può scrivere

$$p_{x_1}(x_2) = \frac{-qp_{x_2}(x_1)}{i\omega\rho A_0 v_{n_0}}$$

dove A_0 è l'area dell'elemento vibrante, v_{n_0} è la velocità normale al punto medio di x_1, ρ è la densità del mezzo e q è l'intensità della sorgente posizionata a x_2. Cioè esiste una semplice relazione algebrica tra la pressione sonora $p_{x_1}(x2)$ causata da un'eccitazione al punto x_1 con la pressione sonora p_{x_2} causata da un'eccitazione al punto x_2.

Al posto di effettuare una simulazione per ogni punto di interesse è sufficiente una singola simulazione invertendo la sorgente con il ricevitore: nel nostro caso metteremo quindi la sorgente (altoparlante) nell'orecchio e il ricevitore (microfono) nei diversi punti di interesse dello spazio.

Capitolo 2

Setup

In questo capitolo viene introdotto il setup hardware e software utilizzato per le simulazioni. Viene inoltre descritto uno scenario di prova utile al verificare il corretto funzionamento del programma di simulazione. Infine, vengono esposte considerazioni e misurazioni riguardanti i parametri utilizzati per l'esecuzione parallela.

2.1 AcouSTO

AcouSTO (Acoustic Simulation TOol) è un simulatore acustico che utilizza la tecnica Boundary Element Method (BEM) per risolvere l'equazione integrale di Kirchhoff-Helmholtz. È opensource e può essere scaricato gratuitamente all'indirizzo `http://acousto.sourceforge.net/`.

Per le simulazioni si è utilizzata la versione 1.5 del programma.

La figura 2.1 mostra la struttura del filesystem di *AcouSTO*. Oltre ai file di configurazione degli *autotools* e ad altri file di testo sono presenti le seguenti cartelle:

1. **blender/** — contenente lo script di esportazione delle mesh dal programma di modellazione 3D *Blender*.
2. **doc/** — contenente il manuale utente e la documentazione *doxygen*.
3. **mysql/** — contenente uno script *SQL* per impostare le tabelle in caso di utilizzo di backend *DBMS MySQL* per il salvataggio dei risultati delle simulazioni.
4. **php/** — contenente un insieme di script *PHP* che permettono di interfacciarsi tramite browser con i dati salvati in *MySQL*.
5. **src/** — contenente il sorgente vero e proprio di *AcouSTO*.

AcouSTO è suddiviso in moduli, ciascuno racchiuso in un file `.c` diverso. I più importanti sono:

- `ac_coef_body.c` — file di implementazione del precalcolo dei coefficienti integrali della superficie.

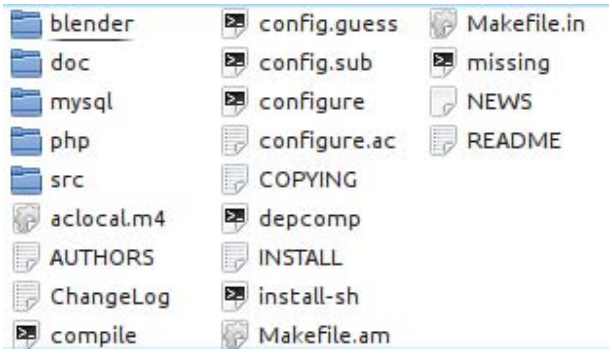


Figura 2.1: *Gerarchia dei file sorgente di AcouSTO.*

- `ac_coef_mics.c` — file di implementazione del precalcolo dei coefficienti integrali dei microfoni.
- `ac_gmres.c` — file di implementazione del risolutore iterativo di sistemi lineari.
- `geom.c` — modulo geometrico.
- `geom_utils.c` — file di implementazione delle funzioni relative alla generazione delle geometrie definite nel file di configurazione.
- `main.c` — core di *AcouSTO*. Si occupa di inizializzare la libreria *MPI*, leggere il file di configurazione, richiamare i moduli necessari e terminare correttamente l'esecuzione.
- `mysqlsave.c` — modulo di interfacciamento con il database *MySQL*.
- `nrwash.c` — file di implementazione delle routine che calcolano il campo incidente e le condizioni al contorno.
- `solution.c` — modulo che si occupa della risoluzione del problema.
- `vtkout.c` — modulo di output della soluzione in formato *Visualization ToolKit* (vtk) e della geometria in formato VRML2. I file generati sono molto utili per la visualizzazione dei risultati in programmi come *paraview*.

2.1.1 Caratteristiche

AcouSTO presenta molte funzionalità interessanti, tra le quali:

1. Gestione delle simmetrie per ridurre il tempo di calcolo
2. Esecuzione parallela su cluster MPI2
3. Sistema di gestione della memoria che consente di controllare la quantità di RAM allocata
4. Gestione di modelli 3D di geometria arbitraria

5. Supporto del DBMS MySQL
6. Nessun limite software sul numero di onde incidenti o sorgenti puntiformi
7. Possibilità di eseguire simulazioni si all'interno che all'esterno di un modello 3D.

I problemi che esso può risolvere sono di due tipi:

1. Scattering di onde piane o sferiche dovuto a molteplici corpi di forma arbitraria
2. Irraggiamento di superfici chiuse vibranti

Per svolgere le simulazioni esso si avvale di librerie standard e ampiamente testate quali *ScaLAPACK*, *BLAS* e *MPI-2*.

2.1.2 Installazione in ambiente Ubuntu Linux

Come prerequisito è necessario scaricare il sorgente dall'indirizzo `http://sourceforge.net/projects/acousto/files/acousto/1.5/acousto-1.5.tar.gz/download`.

A causa del build system personalizzato utilizzato in *AcouSTO*, prima di procedere alla compilazione dei sorgenti è necessario eseguire alcuni passi:

1. Installazione dei pacchetti necessari:

```
> sudo apt-get install gfortran
> sudo apt-get install libconfig8 libconfig8-dev
> sudo apt-get install openmpi-bin
> sudo apt-get install libblacs-mpi1 libblacs-mpi1-dev
> sudo apt-get install libscalapack-mpi1
libscalapack-mpi1-dev
```

2. Creazione dei symlink necessari affinchè *AcouSTO* rilevi la libreria `scalapack`:

```
> cd /usr/lib
> sudo ln -s libscalapack-openmpi.a libscalapack.a
```

   In caso di sistemi diversi da Ubuntu o di distribuzioni *ScaLAPACK* con nomenclatura differente i comandi precedenti vanno modificati adeguatamente.

3. Compilazione:

```
> ./configure
> make
> sudo make install
```

   In caso di errori derivanti da versioni datate degli `autotools` è utile eseguire il comando `autoreconf` prima del passo precedente. Questo programma esegue gli *autotools* ripetutamente per aggiornare il *build system* nella directory corrente e nelle sotto-directory in maniera ricorsiva.

2.1.3 Complessità

La complessità computazionale dell'algoritmo di *AcouSTO* è $O(n^3)$, dove n è il numero di facce di cui è composta la mesh in input. Occorre quindi tenere conto anche del tempo di esecuzione totale della simulazione oltre che dell'utilizzo di memoria RAM.

Un possibile compromesso tra tempo di calcolo e memoria richiesta è dato dal parametro `pre_calculated_coefs`: se viene impostato a 0 i coefficienti integrali vengono calcolati e scartati di volta in volta, mentre se impostato a 1 essi vengono precalcolati e tenuti in memoria.

Ne risulta che nel caso in cui un coefficiente venga riutilizzato più di una volta il fatto di averlo precalcolato porta ad un leggero miglioramento delle prestazioni. Tuttavia, come indicato in [9, par. 4.5.2], questa eventualità non è molto frequente ed è quindi consigliabile lasciare `pre_calculated_coefs` a 0.

È anche importante notare che nelle versioni più recenti di *AcouSTO* sono presenti due risolutori. Il primo, chiamato *PSEUDOINV*, è il risolutore di sistemi lineari integrato nella libreria *ScaLAPACK* e invocato dal file `linsys.c`, mentre il secondo, *GMRES*, è un risolutore iterativo sviluppato da zero dagli autori del programma ed implementato nel file `ac_gmres.c`. Sebbene *GMRES* non sia stato testato sufficientemente, questo tipo di risolutori fornisce delle alte prestazioni di calcolo parallelo con matrici dense, che sono proprio il tipo di matrici con cui lavora *AcouSTO*.

Simmetrie	**Precalcolo = 1**	**Precalcolo = 0**	**Precalcolo = 1**	**Precalcolo = 0**
	GMRES	**GMRES**	**PSEUDOINV**	**PSEUDOINV**
0	1070 GB	152 GB	1070 GB	152 GB
1	268 GB	38 GB	268 GB	38 GB

Tabella 2.1: *Memoria RAM utilizzata al variare di `pre_calculated_coefs` e del tipo di risolutore.*

La tabella 2.1 mostra, per una simulazione di dimensioni significative, come la scelta del tipo di risolutare non influenzi l'utilizzo di RAM, mentre il precalcolo dei coefficienti integrali causa una sostanziale differenza in termini di occupazione di memoria. Viene analizzato solo il caso con una singola simmetria in quanto, come si vedrà nel capitolo 4, è l'unica situazione di interesse nella simulazione di una testa umana.

2.2 Simulazione di prova

Prima di eseguire delle simulazioni onerose è stata creata una configurazione molto semplice che sfrutta le simmetrie della sfera per ridurre drasticamente i tempi di calcolo. In questo modo è stato possibile verificare il corretto funzionamento di *AcouSTO* sull'hardware a disposizione.

2.2.1 File di configurazione

Il file di configurazione di *AcouSTO* è un file di testo letto tramite la libreria `libconfig` ed è diviso in sezioni che si riferiscono ai diversi moduli del programma. Le principali sono:

1. **runinfo** — contenente parametri base della simulazione come nome del run, proprietario, flag di debugging, numero di simmetrie e valori riguardanti l'esecuzione parallela.

2. **modgeom** — dedicato alla descrizione geometrica del problema. Contiene una lista di tutti gli oggetti presenti nell'ambiente 3D. Inoltre è possibile definire il nome del file contenente le coordinate dei microfoni ed il loro numero. Per ogni elemento definito nella lista menzionata precedentemente, viene aggiunta una sezione con il nome corrispondente all'oggetto. Tale sezione contiene la sua definizione ed eventuali operazioni di rototraslazione o scalamento. *AcouSTO* può importare mesh esportate tramite l'apposito script *Blender* oppure creare direttamente sfere, cilindri e piani.

3. **modcoefac** — se attivo assieme al parametro *pre_calculated_coefs* impone di precalcolare i coefficienti integrali relativi alle superfici.

4. **modcoemic** — se attivo assieme al parametro *pre_calculated_coefs* impone di precalcolare i coefficienti integrali relativi ai microfoni.

5. **modsol** — definisce i parametri relativi alla risoluzione del problema. In particolare è possibile impostare il numero di sorgenti e il file contenente le loro coordinate, l'impedenza reale ed immaginaria delle superfici, il tipo di condizione al contorno, la frequenza di partenza e termine della simulazione, il numero di frequenze da simulare, il precalcolo dei coefficienti integrali e l'indice dei microfoni/punti di controllo di cui generare la risposta in frequenza.

Per descrivere i microfoni viene usato un file di testo contenente le coordinate cartesiane degli stessi secondo il seguente formato:

$$\begin{gathered} x_1\ y_1\ z_1 \\ x_2\ y_2\ z_2 \\ \dots \\ x_n\ y_n\ z_n \end{gathered}$$

In modo analogo, per le sorgenti viene usato un file di testo contenente le loro coordinate cartesiane seguite dalla componente reale e immaginaria dell'ampiezza dell'onda acustica generata:

$$\begin{gathered} x_1\ y_1\ z_1\ \Re[A_1]\ \Im[A_1] \\ x_2\ y_2\ z_2\ \Re[A_2]\ \Im[A_2] \\ \dots \\ x_m\ y_m\ z_m\ \Re[A_m]\ \Im[A_m] \end{gathered}$$

Tutte le coordinate sono in metri mentre l'ampiezza dell'onda sonora è in Pascal.

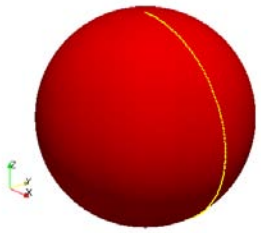


Figura 2.2: *Simulazione di prova: sfera con una sorgente e simmetria assiale.*

2.2.2 Scenario

La simulazione prevede la presenza di:

1. Una sfera di raggio 87.5mm centrata in (0, 0, 0).
2. Una sorgente puntiforme distante 20cm dal centro della sfera di coordinate (0.2, 0, 0).

È importante notare che *AcouSTO* utilizza il metro come unità di misura dello spazio.

L'output della simulazione consiste nell'ampiezza complessa, in $\mathrm{Pascal} = \mathrm{N/m^2}$, della pressione acustica su una singola faccia del modello 3D. I punti in cui viene calcolata la soluzione sono i centroidi delle facce. La soluzione totale si ottiene dalla somma dell'ampiezza dell'onda incidente e dei contributi dovuti allo scattering sul resto della superficie della sfera.

2.2.3 Gestione delle simmetrie

In *AcouSTO* le simmetrie vengono impostate attraverso il parametro `ksymmi`. Per valori inferiori a 3 si sottointende l'esistenza di piani di simmetria, mentre per valori superiori a 3 si indica la presenza di simmetria assiale.

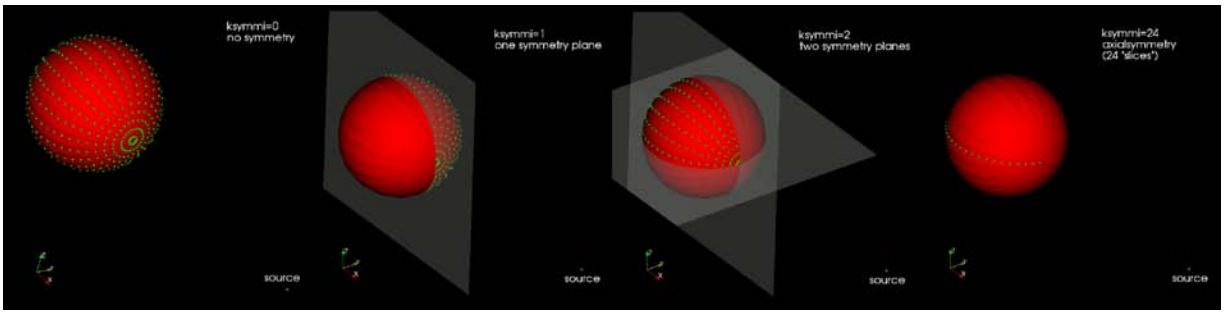


Figura 2.3: *Riduzione dei punti di controllo grazie alle 24 simmetrie di una sfera composta da altrettanti meridiani.*

Ksymmi	Tipo Simmetria	NCNTR/NELEM
0	Nessuna simmetria	1
1	Un piano di simmetria	0.5
2	Due piani di simmetria	0.25
3	Tre piani di simmetria	0.125
>3	Simmetria assiale	1/N

Tabella 2.2: *Effetto di* `ksymmi` *sul numero di punti di controllo.*

La figura 2.3 esplica graficamente l'effetto di `ksymmi`, su una sfera di 24 meridiani, per la sequenza di valori 0, 1, 2, 24.

Il valore di `ksymmi` influenza il rapporto tra elementi della superficie (ovvero le facce) e i punti di controllo, cioè i punti su cui risolvere il sistema di equazioni del BEM:

$$\frac{facce}{punti\ di\ controllo} = \begin{cases} 2^{ksymmi} & ksimmi \leq 3 \\ ksymmi & ksymmi > 3 \end{cases}$$

La tabella 2.2 mostra un resoconto dell'effetto di `ksymmi`.

Come mostrato in seguito, la sfera viene creata con 181 anelli e 362 segmenti. Questo significa che è possibile sfruttare la simmetria assiale con `ksymmi`=362.

Inoltre, affinchè il problema sia completamente simmetrico, la sorgente puntiforme deve essere posizionata lungo l'asse della sfera. I punti di osservazione, o punti di controllo, sono i centroidi distribuiti lungo uno dei meridiani della sfera. La scelta del meridiano non fa differenza a causa della simmetria.

Applicando il principio di inversione del problema acustico, questa simulazione è equivalente a calcolare la *HRTF* di una testa sferica ad una singola distanza.

La figura 2.2 mostra come l'utilizzo delle simmetrie riduca il numero di punti di controllo della sfera, ovvero i punti di cui calcolare la soluzione.

Come indicato nel manuale di *AcouSTO* [9, par. 3.2], sebbene il problema sia simmetrico, è comunque necessaria la presenza della geometria completa.

2.2.4 Sfera

La sfera viene generata direttamente in *AcouSTO*, il quale è dotato di un modulo capace di generare semplici forme geometriche e di applicare alcune trasformazioni come rototraslazione e scalamento.

Il blocco di configurazione necessario è:

```
gui_def_geom={
  type="sphere";
  radius=0.0875;
  segments=362;
  rings=181;
```

```
  translation={x=0.0; y=0.0; z=0.0;};
  rotation={x=0.0; y=0.0; z=0.0;};
};
```

Dove `gui_def_geom` è il nome dato alla sfera creata. Di default la sfera viene creata con l'asse di rotazione coincidente con l'asse z, quindi per preservare la simmetria del problema la sorgente puntiforme va posizionata lungo l'asse stesso, cioè in $(0, 0, 0.2)$.

L'impedenza acustica della superficie della sfera viene impostata a $1+0i$, in modo da simulare una testa acusticamente rigida:

```
radiant_real=1.0;
radiant_imag=0.0;
```

Questa è una buona approsimazione, come mostrato in [6].

2.3 IBM P770

Per le simulazioni è stato utilizzato un sistema *IBM P770 Power7*.

2.3.1 Processore Power7

Il *Power7* è un multiprocessore simmetrico superscalare basato su architettura *PowerPC* a 64 bit.

Il singolo processore *Power7* è dotato di 8 core che lavorano ad una frequenza pari a 3.1GHz. Inoltre, grazie al multithreading simultaneo a 4 vie, ogni core può eseguire fino a 4 thread.

Altra caratteristica degna di nota è la presenza di 32MB di cache di terzo livello.

2.3.2 Configurazione

- 3 drawer[7], ognuno costituito da 2 processori *Power7*, per un totale di $3 \times 2 \times 8 = 48$ core. Il numero massimo di thread eseguibili simultaneamente è quindi $48 \times 4 = 192$.
- 640GB di RAM.
- 16TB di memoria non volatile esterna.

L'organizzazione dei diversi livelli di memoria è illustrata in figura 2.4.

È possibile partizionare RAM e processori in partizioni logiche chiamate *LPAR* (*Logical PARtitions*)[4]. Ogni processore può quindi essere suddiviso in 10 unità logiche, mentre la RAM in blocchi di 256MB. Il sistema può essere riconfigurato dinamicamente a caldo, cioè senza necessità di essere spento o riavviato.

Sono presenti due sistemi operativi differenti che coesistono nella stessa macchina: *AIX6.1* e *SuSE Linux Enterprise 11*. Il primo è un sistema *UNIX* proprietario sviluppato dalla IBM stessa

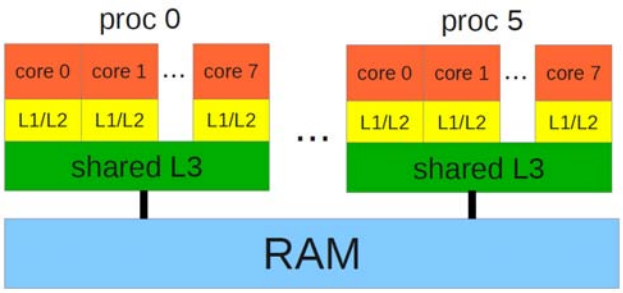


Figura 2.4: *Layout memoria.*

mentre il secondo è una distribuzione *Linux* opensource. Le simulazioni sono state eseguite su quest'ultimo sistema operativo.

Per sfruttare appieno la potenza di calcolo del *P770*, in fase di compilazione di *AcouSTO* è stato utilizzato l'ambiente *IBM Parallel Environment*. Esso non è altro che un insieme di librerie e compilatori, altamente ottimizzati per questa piattaforma hardware, compatibili con gli standard *MPI* e *OpenMP*.

2.3.3 LoadLeveler

LoadLeveler è lo scheduler il cui compito è quello di eseguire i *job* che gli vengono inviati ed allocare le risorse necessarie in base alla classe di esecuzione degli stessi.

Un esempio di *job file* è il seguente:

```
#!/bin/ksh

# @ job_type = parallel
# @ input = /dev/null
# @ error = error.txt
# @ initialdir = /home/smcuser/acousto/multiple_executions/run
# @ output = output.txt
# @ notify_user = smcuser
# @ class = long
# @ notification = error
# @ total_tasks = 81
# @ queue

export PATH=$PATH:/opt/acousto/bin

acousto -f ./configuration.cfg
```

E' facile notare la classe *long* e il numero di processi *total_tasks*.

Il *job* viene inviato a *LoadLeveler* tramite il comando `llsubmit ./job.sh`.

2.4 Performance

Dal punto di vista del carico computazionale, l'algoritmo utilizzato da *AcouSTO* può essere diviso in 3 parti [9, par. 4.6]:

1. Calcolo dei coefficienti integrali
2. Soluzione del sistema lineare
3. Assemblaggio della soluzione nel campo

Il codice sorgente del simulatore contiene chiamate esplicite a funzioni `MPI` per distribuire il calcolo dei coefficienti ai nodi. Questa parte dell'algoritmo è *embarrassingly parallel*, cioè può essere divisa in problemi più piccoli che non necessitano di comunicare tra loro.

Tali sottoproblemi sono creati partizionando in maniera equa il lavoro totale in un numero di problemi pari al numero di processi `MPI`, secondo la tecnica ***SPMD*** (Single Program, Multiple Data) [9, par. 4.6].

2.4.1 ScaLAPACK Data Layout

La libreria *ScaLAPACK* organizza i processi in una griglia. Al fine di rendere la computazione su matrici dense il più veloce possibile, essa utilizza una strategia chiamata *two-dimensional block cyclic distribution* [3, The Two-dimensional Block-Cyclic Distribution.].

Tale configurazione prevede la disposizione dei P processi da eseguire in una griglia $P_r \times P_c$ rettangolare. Le righe e le colonne della stessa vengono divise in blocchi di dimensione MB e NB rispettivamente, poi i dati vengono distribuiti ai processi in maniera ciclica: i dati contenuti all'indice (j, k) della matrice sono salvati nel processo con indice $((j-1)/MB \mod P_r, (k-1)/NB \mod P_c)$.

Ne risulta che una corretta scelta di MB e NB è importante per garantire un buon bilanciamento del carico di dati tra i processi della griglia. MB è detto anche dimensione del blocco di righe mentre NB dimensione del blocco di colonne.

Per le routine utilizzate da *AcouSTO*, la documentazione *ScaLAPACK* consiglia $P_r \ll P_c$ con $NB = MB = 64$ [9, par. 4.6]. Dopo alcuni test informali si è riscontrato che tale configurazione non forniva le migliori performance.

La guida *ScaLAPACK* consiglia griglie di processi quadrate per sistemi a memoria distribuita [3, Achieving High Performance on a Distributed Memory Computer] e si è notato che questa configurazione aumentava la resa del nostro sistema. Si è quindi optato per una griglia quadrata ($P_r = P_c = 9$) per un totale di 81 processi. Il numero di processi è stato dettato dalle risorse messe a disposizione per le simulazioni sul *Power7*.

Al fine di scegliere la dimensione ottimale del blocco di righe e di quello di colonne si è proceduto nella simulazione della sfera per una singola frequenza $f = 10\text{kHz}$ con MB e NB uguali a potenze crescenti di 2 fino a $2^5 = 32$. Il risultato della misurazione dei tempi (in secondi) è esposto nella tabella 2.3.

		column block size					
		1	**2**	**4**	**8**	**16**	**32**
row block size	**1**	8,27	8,66	8,65	9,2	12,54	10,87
	2	8,34	8,3	8,99	10,53	10,87	10,46
	4	8,66	8,27	8,58	10,72	10,86	10,27
	8	10,55	8,72	8,23	9,22	12,68	12,5
	16	10,05	10,03	12,17	14,24	15,9	16,32
	32	10,16	13,84	10,43	12,39	15,01	14

Tabella 2.3: *Tempo di esecuzione al variare di MB (row block size) e NB (column block size), in secondi.*

Per ottenere dei tempi più precisi sarebbe necessario eseguire un numero adeguato di volte il test su ogni singola disposizione dei processi. In questo modo i risultati assumerebbo il significato di andamento medio, risultando quindi molto più attendibili. È tuttavia facile notare come, anche con questi dati preliminari, ci sia una tendenza all'aumento del tempo richiesto all'aumentare di MB e NB.

Ciò potrebbe sembrare in contrasto con quanto consigliato dal manuale di *AcouSTO* e dalla guida di *ScaLAPACK*. Tuttavia, quest'ultima afferma essa stessa che ottenere una configurazione ottima dei parametri citati è un problema non banale [3, Tuning the Distribution Parameters for Better Performance]. È anche vero che, una volta trovato un insieme di buoni parametri, le performance non sono molto sensibili a piccoli cambiamenti dei valori degli stessi.

Capitolo 3

Strumenti sviluppati

3.1 Arresti imprevisti e ripresa della simulazione

AcouSTO calcola ogni frequenza singolarmente e al termine della simulazione passa alla successiva. Ciò significa che in caso di guasti hardware, crash del programma o mancato funzionamento della rete elettrica, il lavoro svolto non viene perduto.

Tuttavia, la capacità di riprendere l'esecuzione da dove si era interrotta non è inclusa e quindi, lanciando semplicemente il programma, la simulazione riprenderebbe dalla prima frequenza indicata nel file di configurazione. Cioè *AcouSTO* ricalcolerebbe anche le frequenze già svolte.

Per risolvere questo problema sono stati sviluppati dei semplici script *python* e *bash* che aggiornano il file di configurazione in modo da riprendere dalla frequenza corretta dopo un'interruzione imprevista.

3.1.1 Strategia adottata

Il normale file di configurazione di *AcouSTO* viene sostituito da un template chiamato `configuration.template` contenente i seguenti *TAG*:

- `NP_GRID_SIZE` — definisce il numero di processi.
- `FREQ_STUFF` — definisce la frequenza di partenza, quella di arresto e il numero di frequenze da simulare.

Prima di lanciare *AcouSTO*, lo script `compile.py` trasforma il template in un file di configurazione vero e proprio.

3.1.2 Implementazione

Per creare un nuovo scenario di simulazione è sufficiente svolgere i seguenti passi:

1. Aprire un terminale e navigare fino alla directory `multiple_execution/`.

2. Copiare la directory `default/` dandole un nome adeguato (in questo esempio `new`) tramite il comando `cp -R default/ new/`.

3. Creare i file necessari alla propria simulazione (es.: microfoni, sorgenti, mesh) come descritto nel paragrafo 2.2.1.

4. Modificare `configuration.template`, facendo attenzione a lasciare commentati i parametri `nprows`, `npcols`, `nome`, `minfreq` e `maxfreq`. Essi sono infatti sostituiti da questi *TAG* che verranno interpretati da `compile.py`:

   (a) `#-# NP_GRID_SIZE <righe> <colonne> #-#` — con `<righe>` e `<colonne>` numeri interi pari rispettivamente all'altezza e alla larghezza della griglia dei processi.

   (b) `#-# FREQ_STUFF <nome> <minfreq> <maxfreq> #-#` — con `<nome>` (numero omega, cioè numero di frequenze da calcolare) intero e `<minfreq>`/`<maxfreq>` float.

Per lanciare lo scenario appena creato:

1. Eseguire `./setup_script.sh ./new/`. Questo copia la cartella `new/` in `run/` aggiungendo anche altri file necessari all'esecuzione (tra cui `compile.py`).

2. Eseguire `./run_script.sh`. Ciò lancia effettivamente la simulazione.

3. In caso di interruzione, imprevista o meno, di *AcouSTO* è sufficiente ripetere il passo 2 per riprendere l'esecuzione da dove era stata interrotta.

Quando la simulazione giunge al termine si può salvare l'output voluto ed eseguire `./delete_run.sh <qualsiasi valore>` per cancellare la cartella `run/`. Il parametro dello script serve solo ad evitare che venga lanciato accidentalmente.

`compile.py`, eseguito tramite `./compile.py <file template> <file di configurazione>`, è il nucleo del meccanismo di ripresa delle simulazioni. Di fatto la sua invocazione viene effettuata da `run_script.sh` in maniera automatica.

La ricerca dei *TAG* da interpretare avviene attraverso le seguenti linee di codice, eseguite su ogni riga del file template in input:

```
1 #Strip leading/trailing whitespaces
2 line = line.strip()
3
4 #We need to compile the given tag
5 if line.startswith("#-#"):
6   #output a simple warning
7   cfg.write("### START OF AUTOGENERATED LINES ###\n")
```

Banalmente, si può notare come venga ricercata la stringa `"#-#"` all'inizio di ogni linea. Dopodiché il *TAG* viene letto e, assieme ai parametri che lo seguono, viene passato alla funzione appropriata:

```
#parse the TAG and its parameters
elements = line.split(" ")
tag = elements[1]
args = elements[2:-1]

#compile the TAG
if elements[1] == "NP_GRID_SIZE":
  nprows, npcols = write_np_grid_size(cfg, args)
elif elements[1] == "FREQ_STUFF":
  write_freq_stuff(cfg, args)
```

Le linee di codice che si occupano della ripresa dall'ultima frequenza calcolata sono le seguenti:

```
def write_freq_stuff(cfg, args):
  """
  Substitute 'FREQ_STUFF nome minfreq maxfreq'.
  """

  #Get the parameters from the template file
  nome = int(args[0])
  minfreq = float(args[1])
  maxfreq = float(args[2])

  if not is_first_execution():
    last_freq = get_last_fresp()

    #If we've already done some calculation we need
    #to find the new 'minfreq' and  'nome'
    if last_freq != 0:
      step = (maxfreq - minfreq) / (nome-1)
      nome = (maxfreq - last_freq)/step
      nome = int(round(nome)) #acousto expect an integer
      minfreq = last_freq + step

  cfg.write("nome="+str(nome)+";\n")
  cfg.write("minfreq="+str(minfreq)+";\n")
  cfg.write("maxfreq="+str(maxfreq)+";\n")
```

AcouSTO calcola il passo di frequenza a partire da `maxfreq`, `minfreq` e `nome`. Quindi, per lasciarlo invariato, si deve aggiornare `nome` in accordo con la nuova frequenza di partenza. Le ultime righe scrivono i valori risultanti nel file di configurazione.

La ricerca dell'ultima frequenza calcolata avviene per mezzo del parsing dei nomi dei file di output nelle cartelle `output/` e `backup/`:

```
def get_last_fresp():
```

```
2  """
3  Get the last calculated frequency. Due to the many outputs
4  AcouSTO can give, we have to search in different paths and
5  extract the frequency information from file's names.
6  """
7
8  files = glob.glob(os.path.join(OUTPUT_DIRECTORY, "*Hz.out"))
9
10 if len(files) == 0:
11   files = glob.glob(os.path.join(BACKUP_DIRECTORY, "*Hz.out"))
12
13 #greatest frequency found so far
14 #(not to be confused with AcouSTO's maxfreq)
15 maxfreq = 0.0
16
17 for f in files:
18   freq = float(f[f.rfind("-")+1 : f.rfind("Hz")])
19   maxfreq = max(freq, maxfreq)
20
21 return maxfreq
```

Nelle ultime righe si può notare come il problema si riduca alla massimizzazione nel valore numerico (in Hertz) presente nel nome dei file.

L'ultimo passo dello script consiste nella scrittura del *jobfile*.

3.2 Affidabilità dei risultati

Il programma di grafica 3D *Blender* fornisce un'interfaccia di scripting molto comoda per estendere le proprie funzionalità. Il calcolo della frequenza massima, come indicato da [5], è implementato mediante una ricerca dello spigolo di lunghezza maggiore tra quelli selezionati. In questo modo viene fornito un limite superiore sulla massima frequenza fino alla quale si possono ottenere dei buoni risultati.

Il filtraggio degli spigoli, in modo da tenere solo quelli selezionati, avviene con queste righe di codice:

```
1 edges = obj.data.edges
2 edges = [ee for ee in edges if ee.select == True]
```

Il fatto di poter selezionare solo una parte della mesh è utile in quanto spesso si vuole che le zone con più dettagli abbiano risoluzione maggiore. Quindi è più sensato misurare la frequenza massima in prossimità di esse.

Il nucleo dello script si riduce alla ricerca dello spigolo di lunghezza massima tra quelli selezionati:

```
1 mm = 0.0
2 for ee in edges:
3     vv = ee.vertices
4     v1 = matrix * vertices[ee.vertices[0]].co
5     v2 = matrix * vertices[ee.vertices[1]].co
6     mm = max((v1-v2).length, mm)
```

È interessante notare come, per ottenere le coordinate reali `v1` e `v2`, sia necessario moltiplicare la matrice `matrix` per le coordinate dei vertici stessi. Infatti, `matrix` è la matrice di trasformazione 4×4 che converte le coordinate dello spazio del modello in quelle dello spazio del "mondo" virtuale (o ambiente 3D globale). Questo passo è necessario perchè la mesh in formato `.obj` fornita dallo scanner 3D utilizza i millimetri come unità di misura base, mentre *Blender* (così come *AcouSTO*) utilizza il metro. Quindi, in questo caso, la moltiplicazione serve per scalare la distanza tra i due estremi dello spigolo in modo da misurare la lunghezza effettiva dello stesso.

Il calcolo della frequenza massima avviene con la formula definita in [5]:

$$f_{max} = \frac{c}{6 \times l_{mm}}$$

dove l_{mm} è lo spigolo di lunghezza massima trovato dalle righe di codice sopra.

Lo script deve essere eseguito all'interno di *Blender*, quindi sono necessari i seguenti passi:

1. Lanciare *Blender* da terminale (altrimenti l'output dello script non sarà visibile).
2. Aprire la mesh desiderata.
3. Dall'editor di testo integrato nel programma, *Text* ▷ *Open Text Block* e selezionare `max_freq.py`.
4. In *Object Mode*, selezionare l'oggetto su cui si vuole fare la misurazione.
5. Entrare in *Edit Mode* e selezionare le zone desiderate della mesh.
6. Tornare in *Object Mode*. Questo passo può sembrare superfluo ma è richiesto affinché *Blender* aggiorni la selezione a cui avrà accesso lo script.
7. Dall'editor di testo, fare click su *Run Script*.
8. Controllare l'output nella schermata del terminale.

In alternativa al passo 2 è possibile installare lo script come plugin di *Blender*, ma visto il campo di utilizzo limitato questa via non è stata seguita.

Elev. (gradi)	-40	-30	-20	-10	0	10	20	30	40	50	60	70	80	90
Passo (gradi)	5	5	5	5	5	5	5	5	5	5	10	15	30	360
Num.	72	72	72	72	72	72	72	72	72	72	36	24	12	1

Tabella 3.1: *Posizione microfoni.*

3.3 Posizionamento microfoni

Il file di *AcouSTO* di definizione dei microfoni è un file di testo contenente su ogni riga le coordinate, separate da uno spazio, degli stessi. La replica della griglia definita da [8] è stata effettuata tramite lo script *mic_mesh.py*.

La tabella 3.1 mostra i valori di azimuth ed elevazione.

Per descrivere i microfoni non si deve far altro che, per ogni raggio voluto, calcolarne le coordinate cartesiane a partire da quelle sferiche verticali polari:

```
1   elev = range(-40, 100, 10)
2   step = [5]*10 + [10, 15, 30, 360]
3   num = [72]*10 + [36, 24, 12, 1]
4
5   print len(elev), len(step), len(num)
6
7   for i,elevation in enumerate(elev):
8     for azimuth in xrange(0, 360, step[i]):
9       az = math.radians(azimuth)
10      el = math.radians(elevation)
11
12      x = distance * math.cos(el) * math.cos(az)
13      y = distance * math.cos(el) * math.sin(az)
14      z = distance * math.sin(el)
15
16      mics.write("%f %f %f\n"%(x, y, z))
```

Lo script va lanciato senza parametri e crea il file *microphones.mesh* da fornire in input ad *AcouSTO*.

La figura 3.1 mostra la griglia completa dei microfoni. Ci sono 793 microfoni per ogni distanza. Le distanze vanno da 20cm a 160cm con passo di 10cm. Questo porta ad avere $[(160 - 20)/10 + 1] \times 793 = 11895$ microfoni in totale.

3.4 Risposta in frequenza dei microfoni

AcouSTO utilizza dei semplici file di testo come output delle simulazioni. Tuttavia, a seconda della configurazione fornitagli, essi possono essere strutturati secondo tre diversi formati:

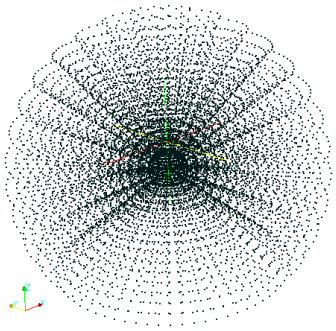


Figura 3.1: *Griglia di 11895 microfoni.*

1. *<nome simulazione>-mics-<freq>Hz.out* e *<nome simulazione>-surf-<freq>Hz.out* — ogni linea contiene una tupla descritta dai seguenti campi: *imic xm ym zm re(inc) im(inc) abs(inc) re(scat) im(scat) abs(scat) re(tot) im(tot) abs(tot).*

   inc è l'ampiezza dell'onda incidente, *scat* è l'ampiezza dei contibuti dovuti allo scattering, *tot* è la somma delle due.

2. *<nome simulazione>-fresp-cnt<cntIndex>.out* — ogni linea contiene una tupla descritta dai seguenti campi: *ifreq omega[rad/s] freq[Hz] re(phi) im(phi) abs(phi).*

3. *<nome simulazione>-fresp-mic<micIndex>.out* — formato simile a quello precedente ma non documentato correttamente.

Dove *ifreq* indica il numero progressivo della frequenza corrente all'interno della simulazione, *imic* è l'indice del microfono a cui si riferisce la riga. I valori numerici sono espressi in $\mathrm{Pascal} = \mathrm{N}/\mathrm{m}^2$.

Il formato che è stato scelto per l'analisi in *matlab* è il secondo, quindi è stato sviluppato uno script python per convertire la risposta in frequenza dei microfoni in modo da non dover distinguere tra i possibili formati di file in fase di post-processing.

A causa dei problemi di interruzione indesiderata delle simulazioni menzionati precedentemente, si è scelto di assemblare le risposte in frequenza dei microfoni a partire dai file relativi all'output delle singole frequenze. L'alternativa sarebbe stata concatenare di volta in volta la risposta in frequenza fornita da *AcouSTO* fino all'interruzione.

Per lanciare lo script si deve innanzitutto spostarsi nella cartella contenente i file *<nome simulazione>-mics-<freq>Hz.out* ed eseguire il comando "`fresp.py <indice primo microfono> <indice ultimo microfono>`".
Se ad esempio la simulazione prevede la presenza di 793 microfoni, come quella del *KEMAR*, il comando da lanciare sarà "`fresp.py 0 792`". Notare che gli indici partono da zero.

Il compito dello script è quello di copiare, per ogni microfono presente nella simulazione, la riga a lui riferita contenuta in ogni file `.out` delle singole frequenze ed incollarla in un unico file `mic<indice>.out`. Eseguendo questo processo seguendo l'ordine crescente delle frequenze, si ottiene il file di risposta in frequenza di tutti i microfoni.

3.5 Simmetrie

Il formato dei file di input per le geometrie, estensione *.nodes*, è molto semplice. Nella prima parte si hanno le coordinate, una per riga e separate da uno spazio, di ogni vertice della mesh. Nella seconda parte ci sono, quattro per riga e separati da uno spazio, gli indici dei vertici che compongono la faccia che si vuole descrivere. Tali indici partono da uno. *AcouSTO* lavora su mesh composte da quadrati e le facce triangolari vengono trattate come quadrati con due vertici coincidenti [9, par. 3.1: BEM grid topology].

Particolare cura deve essere posta nell'ordine in cui gli indici vengono elencati per descrivere una faccia. Ciò influenza la direzione del versore normale utilizzato da *AcouSTO* per distinguere l'interno e l'esterno della superficie.

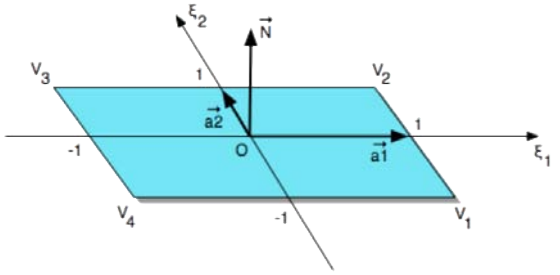


Figura 3.2: *Coordinate curvilinee locali e versore normale.*

Le coordinate curvilinee locali, ξ_1 e ξ_2, sono tali che $\xi_k \in [-1, 1]$, per $k = 1, 2$ (si veda la figura 3.2.

I vettori $\vec{a_1}$, $\vec{a_2}$ ed il versore $\hat{n}$ sono definiti come segue:

$$\vec{a_1} = \frac{\partial x}{\partial \xi_1}, \vec{a_2} = \frac{\partial x}{\partial \xi_2}, \hat{n} = \frac{\vec{a_1} \times \vec{a_2}}{|\vec{a_1} \times \vec{a_2}|}$$

Questi vettori sono valutati numericamente facendo una media delle differenze finite lungo il bordo delle facce. Indicando con $\vec{v_i}$, $i = 1, ..., 4$ la posizione dei 4 vertici si ottiene:

$$\vec{a_1} \simeq \frac{1}{4}(\vec{v_1} - \vec{v_4} + \vec{v_2} - \vec{v_3})$$

$$\vec{a_2} \simeq \frac{1}{4}(\vec{v_2} - \vec{v_1} + \vec{v_3} - \vec{v_4})$$

Da $\vec{a_1}$ e $\vec{a_2}$ si calcola quindi $\hat{n}$.

L'ordine degli indici dei vertici è di conseguenza importante al fine di determinare la direzione del versore normale. La figura 3.3 mostra l'effetto di disporre gli indici in senso orario o antiorario rispetto ad un vertice di partenza.

AcouSTO utilizza un metodo particolare per gestire le simmetrie nel caso di geometrie fornite in input tramite file *.nodes*: la tabella 3.2 mostra gli effetti del parametro *ksymmi* sulla procedura seguita dal programma per sfruttarle.

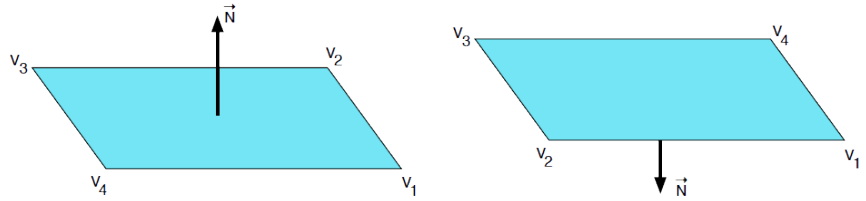


Figura 3.3: *Direzione del versore normale.*

ksymmi	Restrizioni sul file .nodes che descrive la geometria
0	Nessuna
1	La seconda metà di facce deve essere la copia, specchiata rispetto al piano di simmetria, della prima metà.
2	Il secondo quarto di facce deve essere la copia, specchiata rispetto al primo piano di simmetria, del primo quarto. La seconda metà della geometria deve essere la copia, specchiata rispetto al secondo piano di simmetria, della prima metà.
3	Il secondo ottavo di facce deve essere la copia, specchiata rispetto al primo piano di simmetria, del primo ottavo. Il secondo quarto di facce deve essere la copia, specchiata rispetto al secondo piano di simmetria, del primo quarto. La seconda metà della geometria deve essere la copia, specchiata rispetto al terzo piano di simmetria, della prima metà.
> 3	La k-esima fetta della superficie deve essere la copia, ruotata di $(k-1)\frac{2\pi}{ksymmi}$ radianti, delle prime $\frac{n.facce}{ksymmi}$ facce.

Tabella 3.2: *Legame tra simmetrie e contenuto del file nodes.*

Quindi, nel caso in cui lo scenario da simulare possa sfruttare la simmetria della testa, si deve seguire il caso *ksimmy = 1*.

Il risultato è mostrato in figura 3.4.

La strategia utilizzata è quella di esportare da *Blender* un file *.nodes* contenente solo metà della testa. Dopodiché lo script *mirror.py*, il cui compito è quello di ricreare la geometria completa in un altro file, dovrà essere lanciato tramite il comando `./mirror.py <nome file nodes>`.

Aggiungere una copia, con coordinata x opposta, di tutti i vertici contenuti nel file originale può funzionare, tuttavia tutti quelli contenuti nel piano yz verrebbero duplicati.

L'algoritmo utilizzato si può riassumere in pochi passi:

1. Lettura del file di input.
2. Duplicazione di tutti i vertici.
3. Duplicazione delle facce.

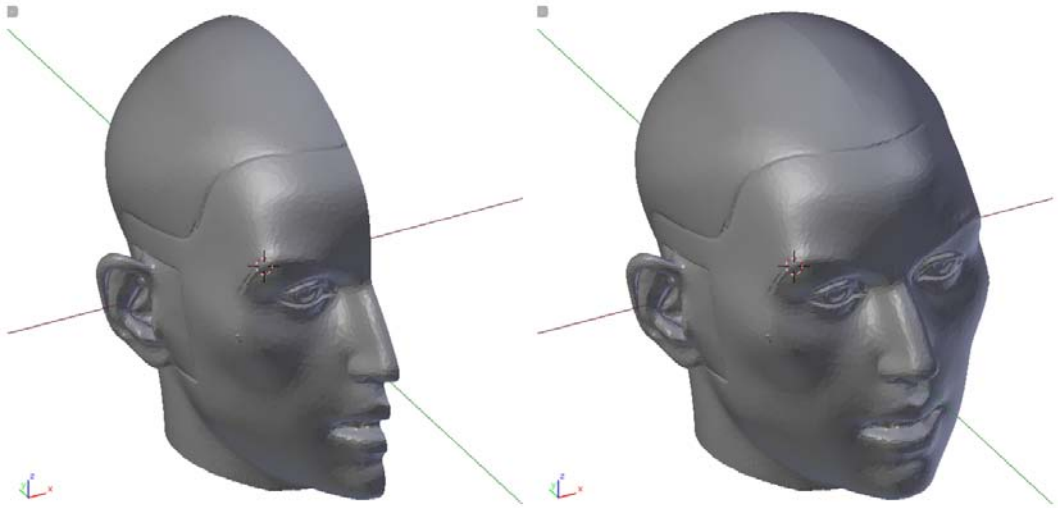


Figura 3.4: *Mesh prima e dopo l'esecuzione di* mirror.py.

4. Rimozione dei vertici doppi

5. Collegamento delle due metà di mesh.

Innanzitutto vengono create le liste *oldv[]* ed *olde[]* che andranno a contenere rispettivamente i vertici e le facce (elementi) contenuti nel file originale. Le liste *v[]* ed *e[]* invece andranno a contenere i vertici e le facce generati per specchiare la mesh. *zero_verts[]* serve per tenere traccia dei vertici che stanno sul piano di simmetria, i quali andranno eliminati.

La duplicazione dei vertici avviene in questa parte di codice:

```
1  #vertices processing
2  lasti = 0
3  lines = open(in_name, "r").readlines()
4  for i,line in enumerate(lines):
5    tokens = [float(x) for x in line.split()]
6
7    if len(tokens) != 3:
8      lasti = i
9      break
10
11   oldv.append(tokens)
12   #if the vertex doesn't lie in the
13   #symmetry plane (0, y, z) mirror it
14   #and add it to the new vertices list
15   if tokens[0] != 0:
16     t = tokens[:]
17     t[0] *= -1
```

```
18      v.append(t)
19    #if it does lie in the plane, add it
20    #to the list of vertices that need
21    #to be deleted
22    else:
23      zero_verts.append(i)
```

Si può notare come le coordinate dei vertici vengano lette riga per riga. Si distingono due casi:

- $x \neq 0$, cioè il vertice non appartiene al piano di simmetria. In questo caso la componente x viene specchiata tramite la moltiplicazione per il fattore -1 e il vertice ottenuto viene inserito alla fine della lista *v[]*.
- $x = 0$, cioè il vertice appartiene al piano di simmetria. In questo caso l'indice del vertice viene aggiunto a *zero_verts[]* per un utilizzo successivo. Le coordinate del vertice, invece, vengono scartate.

Dopodiché le facce originali vengono caricate e salvate in *olde[]*:

```
1  #faces processing
2  for line in lines[lasti:]:
3    tokens = [int(x) for x in line.split()]
4
5    if len(tokens) != 4 and line != "":
6      print "Errore"
7      exit(0)
8
9    olde.append(tokens)
```

Ora si deve ricostruire la faccia specchiata prestando attenzione a due casi che si possono presentare analizzando i vertici che la descrivono:

1. Se il vertice appartiene al piano di simmetria, bisogna riferirsi all'indice originale all'interno di *oldv[]* in quanto non è presente un nuovo vertice a cui riferirsi. Questa è di fatto l'operazione che collega le facce di confine.
2. Se il vertice non appartiene al piano, essendo stati cancellati alcuni vertici, al nuovo indice si deve sottrarre il numero di vertici con $x = 0$ incontrati finora.

Questa è la procedura più delicata ed è eseguita con le seguenti righe di codice:

```
1    #Remove concident vertices
2    #and update the others' indexes
3    newtokens = []
4    for t in tokens:
5      if oldv[t-1][0] == 0:
```

```
6         newtokens.append(t)
7       else:
8         newtokens.append(t+old_nvert-count_zeros(t-1))
```

Di particolare interesse sono le righe:

```
1       if oldv[t-1][0] == 0:
2         newtokens.append(t)
```

Che rappresentano il primo caso tra quelli elencati sopra. L'indice del vertice si riferisce al vertice, contenuto in *oldv[]*, del file originale. Mentre

```
1       else:
2         newtokens.append(t+old_nvert-count_zeros(t-1))
```

è il secondo caso. Il nuovo indice è calcolato come *indice originale + numero di vertici presenti nel file in input - numero di vertici del file originale con x=0 trovati fino al* $(t-1)$*-esimo*. In forma matematica:

$$i_{new} = i_{old} + n_{half} - \sum_{z \in zero\_verts} \delta_{-1}((i_{old} - 1) - z) \tag{3.1}$$

Dove i_{new} è l'indice del nuovo vertice, i_{old} è l'indice del vecchio vertice (cioè il corrispondente di quello nuovo nella prima metà di mesh) e δ_{-1} è la funzione gradino.

L'ordine dei vertici viene poi invertito per invertire la direzione del versore normale facendola puntare verso l'esterno:

```
1   #Reverse new vertices indexes' order to keep
2   #the normal vector pointing outward
3   newtokens.reverse()
4   e.append(newtokens)
```

La funzione *count_zeros(indice)* è implementata nel seguente modo:

```
1  def count_zeros(x):
2    """
3    sum(n < x for n in zero_verts)
4    but somewhat faster
5    """
6    res = 0
7    for n in zero_verts:
8      if n > x:
9        break
10
11     res += 1
12
13   return res
```

In pratica viene eseguita una conta del numero di vertici nel piano di simmetria con indice minore o uguale all'indice dato.

L'algoritmo utilizzato non è molto efficiente ma è concettualmente semplice e, anche su una mezza mesh di 199.924 facce e 100.799 vertici, viene eseguito dall'interprete *python* ottimizzato *pypy* in poco più di un secondo.

Capitolo 4

Simulazione modello mesh KEMAR

Per la simulazione della risposta in frequenza di una testa umana (HRTF) è stato usato il modello 3D di un manichino KEMAR[1] per test acustici appartenente al gruppo di ricerca *SMC* (*Sound Music Computing*) e utilizzato dal team di tecnologie binaurali.

Il modello è stato ottenuto tramite scansione laser di un manichino reale.

4.1 Acquisizione mesh

L'acquisizione della testa del manichino *KEMAR* è stata fatta con lo scanner 3D *NextEngine 3D Scanner HD*[2] visibile in figura 4.1 in dotazione al *Laboratorio di Tecnologia e Telecomunicazioni Multimediali* (*LTTM*).

Figura 4.1: *Scanner 3D.*

4.1.1 Specifiche scanner

La tabella 4.1 mostra le specifiche dello scanner 3D utilizzato per le due possibili modalità di utilizzo *macro* e *wide*.

	Modalità	
	macro	**wide**
Sorgente	Coppia di 4 laser a stato solido classe 1M da 10mW. Lunghezza d'onda pari a 650nm.	
Sensore	Coppia di sensori CMOS da 3.0 Megapixel.	
Dimensione massima oggetti	Nessun limite. Se la dimensione di un oggetto eccede quella del campo di acquisizione possibile effettuare molteplici scansioni parziali e ricomporre la mesh completa tramite l'apposito software.	
Campo d'acquisizione	$129.54\,\mathrm{mm} \times 96.52\,\mathrm{mm}$	$342.9\,\mathrm{mm} \times 256.54\,\mathrm{mm}$
Densità acquisizione geometria	$248\,\mathrm{punti}^2/\mathrm{mm}^2$	$34.9\,\mathrm{punti}^2/\mathrm{mm}^2$
Densità acquisizione texture	$400\,\mathrm{DPI}$	$150\,\mathrm{DPI}$
Accuratezza dimensionale	$0.127\,\mathrm{mm}$	$0.381\,\mathrm{mm}$
Velocità di acquisizione	$50000\,\mathrm{punti/s}$. Tipicamente 2 minuti per scansione di una singola faccia.	
Formati di output	STL, OBJ, VRML, XYZ, U3D e PLY.	

Tabella 4.1: *Specifiche scanner 3D.*

4.1.2 Post-processing

Il modello 3D acquisito necessita di alcune operazioni prima di essere utilizzabile. Inoltre ha una risoluzione molto fine, ben superiore a quanto necessario per eseguire simulazioni nel campo delle frequenze udibili da un essere umano (20Hz-20.000Hz).

È stata seguita la seguente procedura:

1. **Importazione** del file `.obj` in *Blender*.
2. **Ridimensionamento della mesh**, tramite fattore di scala 0.001, per passare da millimetri a metri.
3. **Eliminazione delle imperfezioni** introdotte nell'acquisizione (figura 4.2):
    - In *Edit Mode*, partendo da una selezione nulla, selezionare un vertice della mesh del *KEMAR*. (a, b)

- *Select* ▷ *Linked*. (c, d)
- *Select* ▷ *Inverse*. (e)
- *Mesh* ▷ *Delete...* ▷ *Vertices*. (f, g, h)

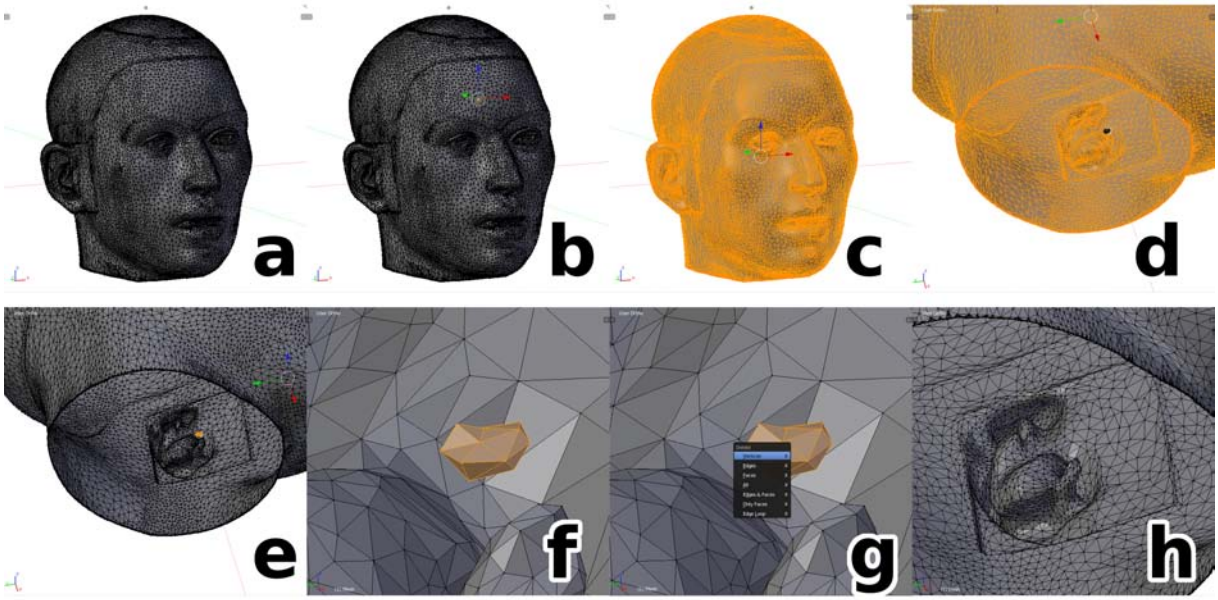


Figura 4.2: *Eliminazione imperfezioni dovute all'acquisizione del manichino.*

4. **Chiusura della superficie** (figura 4.3):

   - In *Edit Mode*, partire da una selezione nulla. (a)
   - *Select* ▷ *Non Manifold*. (b, c)
   - *Mesh* ▷ *Faces* ▷ *Fill*. (d, e)

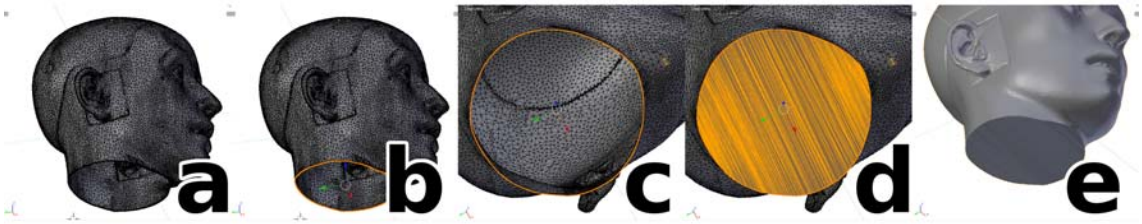


Figura 4.3: *Chiusura del collo del manichino.*

5. **Centramento e rotazione** approssimativa della mesh tenendo come riferimento l'asse X, il quale viene fatto passare attraverso il canale uditivo.

6. **Miglioramento della superficie di chiusura del collo** (figura 4.4). Questo non è un passo necessario ma rimuove molti vertici inutili. L'idea di base è di sottrarre alla mesh della testa un cubo che interseca lievemente il collo. In questo modo si ottiene un taglio preciso nettamente migliore delle facce generate automaticamente da *Blender* nel passo precedente. La procedura è la seguente:

   - Creare un cubo sufficientemente grande da tagliare il collo. (a)
   - Spostare / ruotare il cubo in modo che intersechi completamente la base del collo. È importante prestare attenzione a non intersecare una parte troppo grande della mesh *KEMAR*. (b)
   - In *Object Mode*, con la sola testa selezionata, aprire la scheda *Object Modifiers* del pannello *Properties*. (c)
   - *Add Modifier* ▷ *Boolean*. (c)
   - Selezionare l'operazione *Difference* con il cubo appena creato. (c)
   - Fare click su *Apply*. (c)
   - Cancellare il cubo. (d, e)

Figura 4.4: *Miglioramento della chiusura del collo.*

7. Eventuale **suddivisione delle facce del collo** in modo da ottenere una superficie più fine (figura 4.5):

   - In *Edit Mode*, *Face select mode*, selezionare una delle facce del collo. (a)
   - *Select* ▷ *Similar* ▷ *Co-planar*. (b)
   - Ripetere *Barra Spaziatrice* ▷ *Subdivide* fino ad ottenere un risultato adeguato. (c, d)

Se non si vuole sfruttare la simmetria della testa non resta che esportare la mesh tramite l'apposito plugin presente nella directory `blender/` dei sorgenti di *AcouSTO* per poterla poi importare nel simulatore. In caso contrario è necessario eseguire dei passi aggiuntivi.

La mesh non è perfettamente simmetrica a causa delle imperfezioni che affliggono oggetti reali e di quelle introdotte dal processo di acquisizione da parte dello scanner 3D. Quindi è necessario dividere in due metà la testa e cancellarne una. Il modello completo viene poi ricostruito da uno script sviluppato appositamente e descritto al paragrafo 3.5.

La procedura da seguire per fare ciò è la seguente:

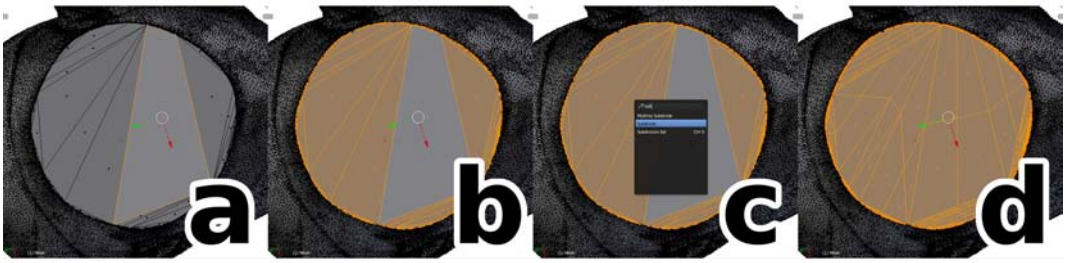


Figura 4.5: *Suddivisione delle facce del collo.*

1. **Centrare e ruotare** la testa in modo che il piano contenente gli assi x e y coincida con il piano di simmetria della testa.

2. **Creare un cubo** di dimensione sufficiente a contenere metà della testa. (a)

3. **Posizionare il cubo** appena creato in modo da sovrapporlo alla metà della testa che si vuole cancellare. (b)

4. **Sottrarre il cubo** dalla mesh della testa:

   - In *Object Mode* selezionare la testa.
   - Nel pannello *Modifiers* selezionare *Add Modifier* ▷ *Boolean*.
   - Selezionare *Operation* ▷ *Difference*.
   - Impostare il cubo come oggetto da sottrarre.
   - Cancellare il cubo

5. **Cancellare le facce** che *Blender* ha aggiunto per chiudere la testa tagliata:

   - Selezionare la testa ed entrare in *Edit Mode*.
   - Scegliere la *Face select mode*.
   - Selezionare una faccia di quelle da rimuovere. (c)
   - *Select* ▷ *Similar* ▷ *Co-planar*. (d)
   - *X* ▷ *Faces*. (e)

La sequenza di operazioni è illustrata nella figura 4.6.

Per la simulazione del *KEMAR* la simmetria della testa non è stata sfruttata in quanto, sebbene sia presente una simmetria geometrica del modello, la posizione di sorgente/microfoni rende il problema non simmetrico. L'algoritmo di *AcouSTO* non è stato formulato per gestire questo tipo di configurazioni.

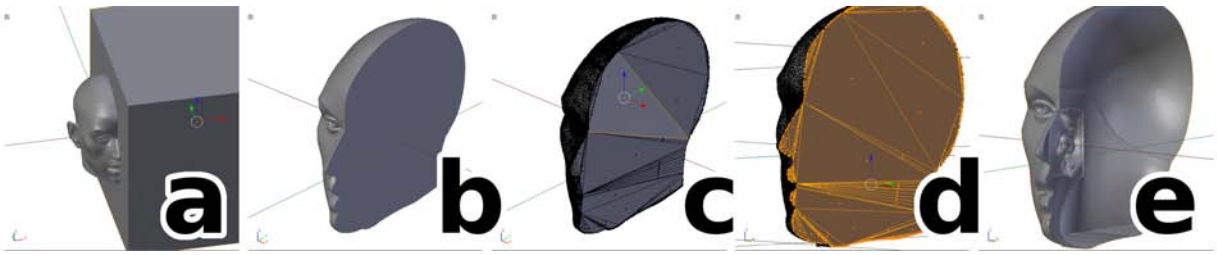


Figura 4.6: *Procedura per cancellare metà testa.*

4.2 Scenario

La simulazione prevede la presenza di una sorgente puntiforme nell'orecchio destro (4.7) e di una griglia di 793 microfoni disposti a 160cm dal centro della testa. Il passo di azimut/elevazione segue la tabella 4.2 come descritto in [8]. Il tutto è rappresentato in figura 4.8.

La sorgente ha coordinate, in metri, $(-0.06698, 0.000288, 0.0)$ e la sua distanza minima dalla testa è di circa $1.024\,\mathrm{mm}$.

Figura 4.7: *Sorgente puntiforme nell'orecchio destro del manichino.*

Il sistema di riferimento di azimut ed elevazione differisce da quello usato in [8] (figura 4.9) ed è mostrato in figura 4.10. Guardando la testa dall'alto, l'azimut si riferisce all'angolo antiorario tra l'orecchio sinistro (visto dal manichino). Esso varia da 0 a 360.

Per quanto riguarda l'elevazione, si riferisce al piano orizzontale. Viene fatta variare tra -40 e 90 a passi di 10 gradi.

Si tratta quindi di un sistema polare-verticale con $\theta = [0, 360)$ e $\phi = [-40, 90]$.

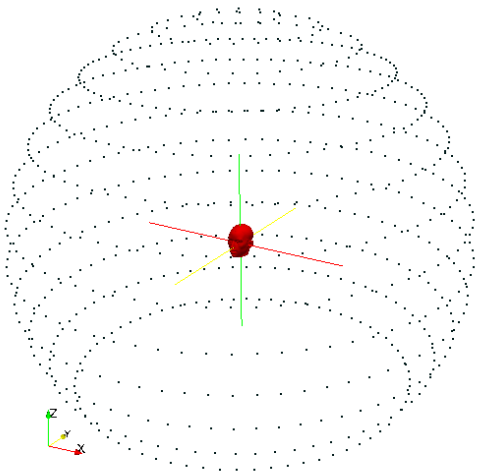


Figura 4.8: *Griglia di 793 microfoni.*

Elev. (gradi)	-40	-30	-20	-10	0	10	20	30	40	50	60	70	80	90
Passo (gradi)	5	5	5	5	5	5	5	5	5	5	10	15	30	360
Num.	72	72	72	72	72	72	72	72	72	72	36	24	12	1

Tabella 4.2: *Posizione microfoni.*

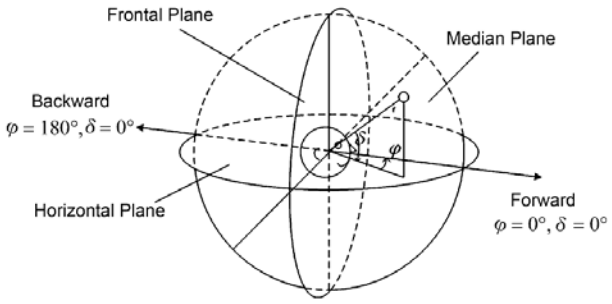


Figura 4.9: *Coordinate sferiche verticali polari usate da [8] per i microfoni. Azimut ϕ ed elevazione δ.*

4.3 Benchmark

La mesh utilizzata, dopo un numero di decimazioni tali da rendere il problema trattabile in termini di tempo, è composta da 21.128 vertici per un totale di 24.898 facce. La stima sulla memoria necessaria fornita in output da *AcouSTO* (parametro `-m`) ammonta a 9,3GB. Il tutto porta ad un tempo di calcolo di circa 1 ora per frequenza.

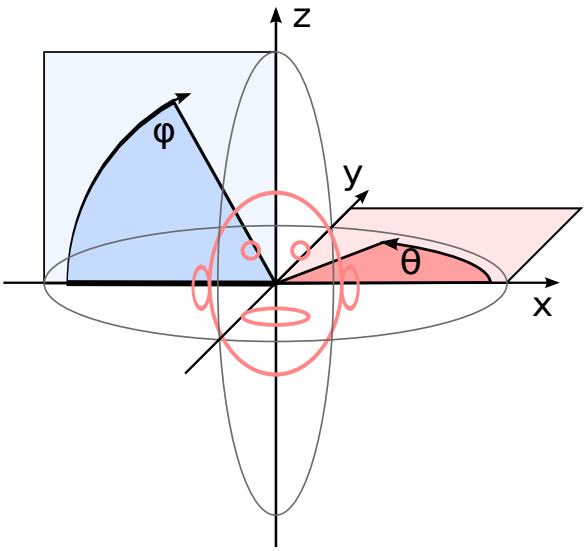


Figura 4.10: *Coordinate sferiche verticali polari usate per i microfoni. Azimut ϕ ed elevazione θ.*

4.3.1 Decimazione mesh

Al fine di ottenere dei modelli trattabili la mesh acquisita dallo scanner è stata decimata più volte. Questa procedura dimezza, all'incirca, il numero di vertici che descrivono la superficie. Per farlo è stato utilizzato il *modifier* **Decimate** incluso in *Blender*.

La figura 4.11 mostra come viene modificata la mesh ad ogni decimazione. La figura 4.12 mostra il dettaglio della zona di maggior interesse, cioè l'orecchio. In entrambi i casi, il numero di facce parte dal valore 1.341.232 e viene circa dimezzato ad ogni passaggio fino ad arrivare al valore 654.

Si può notare come l'algoritmo di *Blender* mantenga più definite le aree che richiedono una maggiore precisione per essere descritte adeguatamente.

La tabella 4.3 mostra la memoria stimata per diverse risoluzioni della mesh del manichino *KEMAR*. Il numero di simmetrie è 0.

n. facce	n. vertici	RAM (GB)
1341476	670714	6700
167654	83833	418
83826	41919	105
41912	20962	26
24898	21128	9,3

Tabella 4.3: *Memoria stimata per diverse risoluzioni della mesh.*

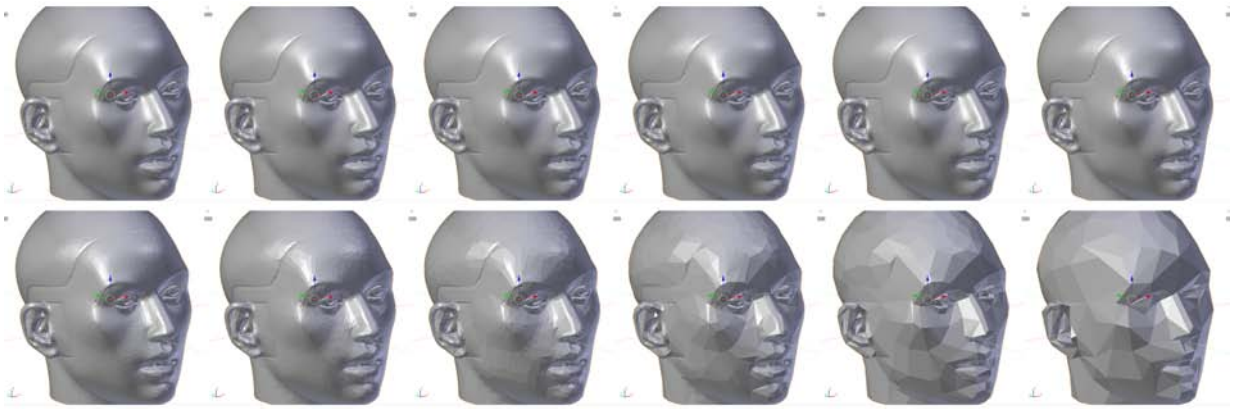


Figura 4.11: *Decimazione: testa.*

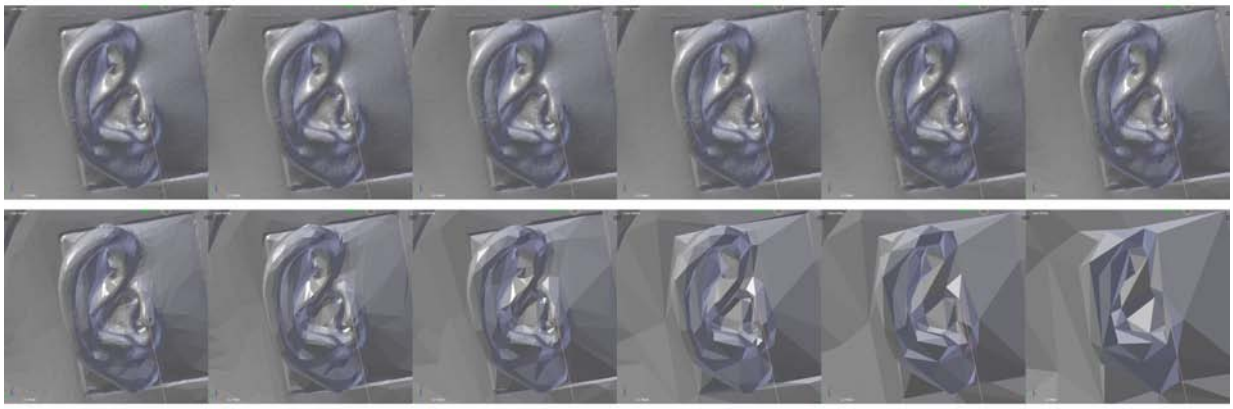


Figura 4.12: *Decimazione: dettaglio orecchio.*

4.3.2 Frequenza massima

Per il calcolo della frequenza massima è stata utilizzata la formula fornita da Katz in [5], nelle due versioni:

$$f^6_{max} = \frac{c}{6 \times spigolo_{max}} \tag{4.1}$$

$$f^4_{max} = \frac{c}{4 \times spigolo_{max}} \tag{4.2}$$

dove $c = 343\ m/s$ è la velocità del suono e

$$spigolo_{max} = \max_{s \in spigoli} lunghezza(s)$$

è la misura dello spigolo più lungo.

La prima formula rappresenta il fatto che almeno 6 elementi del contorno sono necessari a rappresentare accuratamente il periodo di un'onda sonora, mentre 4 è il numero minimo.

La tabella 4.4 mostra il valore di f_{max}, a sei o quattro elementi di contorno, per ogni passo di decimazione mostrato nelle figure 4.11 e 4.12.

Decimazioni	**Facce**	f^6_{max}	f^4_{max}
0	1341232	23935	35902
1	670616	23551	35326
2	335308	22339	33508
3	167654	16023	24035
4	83826	11550	17325
5	41912	9463	14195
6	20956	8578	12867
7	10478	7000	10500
8	5238	6528	9791
9	2618	5456	8184
10	1308	4087	6131
11	654	3493	5239

Tabella 4.4: *Relazione tra numero di decimazioni e* f_{max}

Tramite misurazioni eseguite sull'orecchio della mesh, risulta che lo spigolo di lunghezza maggiore è lungo 9,023mm. Utilizzando la formula 4.1 otteniamo 6.335Hz, mentre con la 4.2 otteniamo 9.503Hz. Questo significa che fino a poco più di 6kHz i risultati dovrebbero essere ottimi, mentre avvicinandosi ai 9,5kHz e superandoli la garanzia sulla loro qualità viene meno.

La simulazione è stata comunque effettuata fino a 20kHz per analizzare il deterioramento dell'output.

4.4 Risultati

La figura 4.13 mostra un grafico tridimensionale delle HRTFs ad elevazione 0° utilizzando come sistema di riferimento quello utilizzato da [8]. È importante sottolineare come a 90° di azimut ci sia una zona di alta pressione acustica mentre a 270° sia presente una zona di bassa pressione acustica. Ciò corrisponde, rispettivamente, alle coordinate dell'orecchio destro e di quello sinistro. Nel primo infatti è posizionata la sorgente mentre nel secondo la testa crea un cono d'ombra che attenua il segnale.

La figura 4.14 mostra un confronto tra le HRTFs simulate, a sinistra, e quelle presenti in letteratura, a destra, per elevazione 0° e azimut 0°.

Come si può notare i due grafici non mostrano un'ottima corrispondenza, tuttavia si osserva che i principali notch delle risposte coincidono a basse frequenze. È utile sottolineare anche che l'output della simulazione per frequenze superiori a 6kHz non è attendibile a causa della forte decimazione della mesh utilizzata.

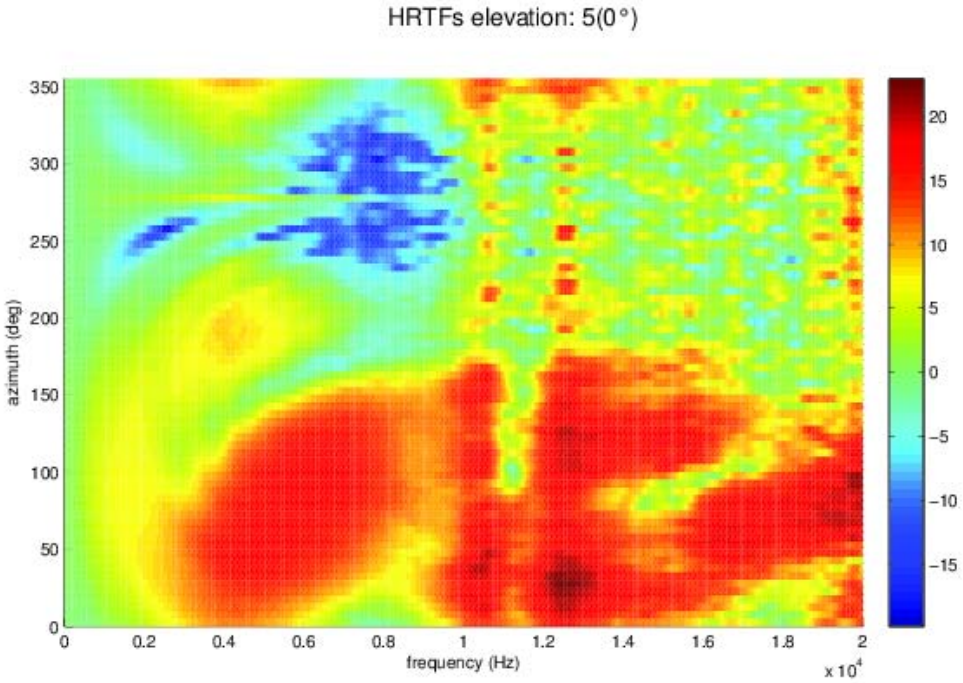
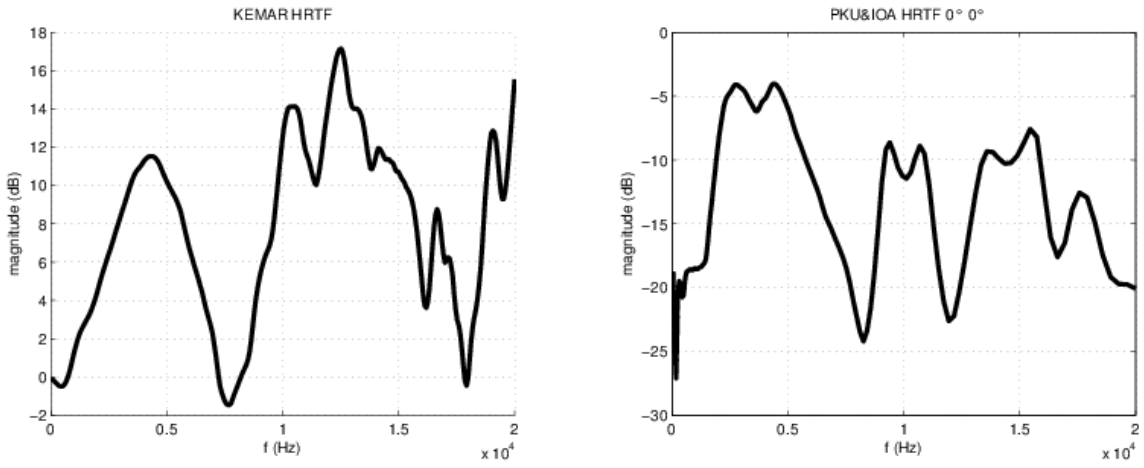


Figura 4.13: *3D-HRTFs*



(a) HRTF simulata (b) HRTF PKU&IOA

Figura 4.14: *Elevazione 0, Azimuth 0*

Capitolo 5

Conclusioni

Questo lavoro di tesi ha avuto come obiettivi: l'installazione e la configurazione di *AcouSTO* sul *Power7*, la simulazione di *HRTF* per una testa di manichino *KEMAR* presente nel laboratorio di Informatica Musicale del gruppo di ricerca *Sound & Music Computing Group* e la risoluzione di tutte le problematiche riscontrate nel farlo per rendere i risultati delle simulazioni numeriche attendibili.

Il notevole tempo richiesto per eseguire e portare a termine simulazioni non banali, come quella sulla mesh del manichino *KEMAR*, e i problemi di compatibilità con il *Power7*, hanno reso al momento impraticabile l'utilizzo di *AcouSTO* nell'ottenere risultati significativi su tutto l'intervallo di frequenze udibili da un essere umano. Il principale vincolo alla simulazione del manichino è stato il tempo impiegato per la soluzione di una singola frequenza. La memoria RAM utilizzata è solo una piccola parte di quella disponibile in quanto, aumentando la risoluzione della mesh per utilizzarla nella sua totalità, si sono ottenuti tempi di esecuzione dell'ordine dei mesi.

Le tecniche di post-processing sviluppate sono risultate efficaci nell'eliminazione dei difetti di acquisizione della mesh, inoltre la possibilità di gestire una simmetria permette di ottenere i vantaggi di velocità e utilizzo RAM di una decimazione senza perdita di definizione del modello.

Gli script dedicati all'inizializzazione e all'esecuzione e ripristino delle simulazioni si sono dimostrati tool indispensabili per gestire le varie configurazioni e ridurre il tempo di creazione di nuovi scenari.

Infine, i vari benchmark hanno sottolineato come sia possibile utilizzare *AcouSTO* per effettuare simulazioni di interesse scientifico e pratico. Il limite principale resta quello delle risorse computazionali a disposizione, tuttavia esse sono destinate ad aumentare con il continuo progresso tecnologico. Allo stesso modo, il progresso nello sviluppo del software di simulazione, soprattutto per quanto concerne il risolutore di sistemi lineari *GMRES*, promette una riduzione dei tempi di esecuzione, che sono il principale vincolo anche nel caso del manichino *KEMAR*.

5.1 Sviluppi futuri

Tra i numerosi sviluppi futuri del lavoro svolto vengono riportati quelli di più immediata realizzazione:

- **Utilizzo di modelli 3D non uniformi:** la non uniformità della mesh permetterebbe di mantenere alta la risoluzione nelle zone ricche di dettagli, fondamentali per ottenere una risposta in frequenza accurata, e abbassare la risoluzione nelle altre. In questo modo il tempo di calcolo e la quantità di memoria RAM richiesta diminuirebbero significativamente senza andare a influenzare troppo la qualità dei risultati.
- **Inclusione di un modello mesh del torso:** In letteratura è riportato che la diffrazione dovuta al torso è un contributo apprezzabile nella HRTF soprattutto come indicatore di localizzazione verticale. Per questo motivo è utile includerlo in future simulazioni acustiche. È anche interessante valutare il solo contributo del busto in assenza della testa.
- **Modelli mesh di persone:** Sebbene l'utilizzo del modello di un manichino fornisca dei risultati interessanti anche a fini pratici, è comunque importante valutare come le caratteristiche antropomorfiche possano influire sulla HRTF.
- **Utilizzo di un cluster più potente:** consente l'esecuzione di simulazioni più dettagliate. Di particolare interesse è la possibilità di integrare in *AcouSTO* tecnologie *GPGPU*, come *Nvidia CUDA* o *OpenCL*, per sfruttare l'alto parallellismo delle schede grafiche ed incrementare significativamente la potenza di calcolo disponibile.
- ***Virtual Acoustics***: simulazione dell'acustica di interi ambienti virtuali quali modelli di stanze, case o simili.

Appendici

Appendice A

Comandi

In questo appendice sono descritti i comandi da lanciare per inizializzare, riprendere e terminare le simulazioni.

A.1 Inizializzazione

Innanzitutto vanno creati i file necessari a descrivere lo scenario che si desidera simulare. Questi file vanno posti in una sotto-directory di `multiple_executions/`, dando un nome significativo alla cartella.

Dopodiché, da un terminale aperto in `multiple_executions/`, si deve lanciare il comando `./setup_script.sh <nome cartella scenario>`. Questo copia la directory in `run/`, dove verrà eseguita la simulazione effettiva, e crea dei file aggiuntivi necessari al meccanismo di ripresa dell'esecuzione per funzionare.

A.2 Inizio, arresto e ripresa dell'esecuzione

Per lanciare la simulazione è sufficiente utilizzare il comando `./run_script.sh`. Per arrestarla si utilizza `./stop_script.sh`.

In caso di interruzione, facilmente verificabile tramite il comando `llq`, per riprendere l'esecuzione si utilizza di nuovo il comando `./run_script.sh`.

A.3 Termine simulazione

È possibile verificare l'ultima frequenza simulata tramite il comando `./last_freq.sh`.

Una volta terminata correttamente la simulazione, l'esecuzione del comando `./finished.sh` si occupa di spostare i file `.out` nelle cartelle `out/mics/` e `out/surf/` in base al tipo di output.

A.4 Cleanup

Per ripulire la directory `run/` da ogni file relativo alla simulazione, compresi i dati di output, si utilizza lo script `./delete_run.sh <qualsiasi valore>`. Il parametro dello script è ignorato ed serve solo da protezione contro invocazioni involontarie del comando.

Appendice B

Codice

B.1 compile.py

```
1  #!/usr/bin/env python
2
3  import sys, os
4  import glob
5
6  #Configurazione
7  BACKUP_DIRECTORY = "./backup"
8  OUTPUT_DIRECTORY = "./out"
9  JOB_SH = "job.sh"
10
11 def write_np_grid_size(cfg, args):
12   """
13   Substitute 'NP_GRID_SIZE nprows npcols'.
14   """
15
16   nprows = int(args[0])
17   npcols = int(args[1])
18   cfg.write("nprows="+str(nprows)+";\n")
19   cfg.write("npcols="+str(npcols)+";\n")
20
21   return nprows, npcols
22
23 def write_freq_stuff(cfg, args):
24   """
25   Substitute 'FREQ_STUFF nome minfreq maxfreq'.
26   """
27
```

```
28    #Get the parameters from the template file
29    nome = int(args[0])
30    minfreq = float(args[1])
31    maxfreq = float(args[2])
32
33    if not is_first_execution():
34      last_freq = get_last_fresp()
35
36      #If we've already done some calculation we need
37      #to find the new 'minfreq' and  'nome'
38      if last_freq != 0:
39        step = (maxfreq - minfreq) / (nome-1)
40        nome = (maxfreq - last_freq)/step
41        nome = int(round(nome)) #acousto expect an integer
42        minfreq = last_freq + step
43
44    cfg.write("nome="+str(nome)+";\n")
45    cfg.write("minfreq="+str(minfreq)+";\n")
46    cfg.write("maxfreq="+str(maxfreq)+";\n")
47
48  def get_last_fresp():
49    """
50    Get the last calculated frequency. Due to the many outputs
51    AcouSTO can give, we have to search in different paths and
52    extract the frequency information from file's names.
53    """
54
55    files = glob.glob(os.path.join(OUTPUT_DIRECTORY, "*Hz.out"))
56
57    if len(files) == 0:
58      files = glob.glob(os.path.join(BACKUP_DIRECTORY, "*Hz.out"))
59
60    #greatest frequency found so far
61    #(not to be confused with AcouSTO's maxfreq)
62    maxfreq = 0.0
63
64    for f in files:
65      freq = float(f[f.rfind("-")+1 : f.rfind("Hz")])
66      maxfreq = max(freq, maxfreq)
67
68    return maxfreq
69
70  def compile(tmpl_name, cfg_name):
```

```
71    """
72    Read the given template file line by line
73    to search for known TAGs.
74    """
75
76    tmpl = open(tmpl_name, "r").readlines()
77    cfg = open(cfg_name, "w")
78
79    nprows = 0
80    npcols = 0
81
82    for line in tmpl:
83      #Strip leading/trailing whitespaces
84      line = line.strip()
85
86      #We need to compile the given tag
87      if line.startswith("#-#"):
88        #output a simple warning
89        cfg.write("### START OF AUTOGENERATED LINES ###\n")
90
91        #parse the TAG and its parameters
92        elements = line.split(" ")
93        tag = elements[1]
94        args = elements[2:-1]
95
96        #compile the TAG
97        if elements[1] == "NP_GRID_SIZE":
98          nprows, npcols = write_np_grid_size(cfg, args)
99        elif elements[1] == "FREQ_STUFF":
100         write_freq_stuff(cfg, args)
101
102       cfg.write("### END OF AUTOGENERATED LINES ###\n")
103     else:
104       #if we are on a "normal" line, copy/paste it
105       cfg.write(line + "\n")
106
107   cfg.close()
108
109   create_run_file(cfg_name, nprows, npcols)
110
111 def create_run_file(cfg_path, nprows, npcols):
112   """
113   """
```

```
114
115   #Single machine execution
116   script = """#!/bin/sh
117 # %s
118 mpirun acousto -np %i -f %s
119 """
120
121   #Power7 execution
122   script = """#!/bin/ksh
123
124 # @ job_type = parallel
125 # @ input = /dev/null
126 # @ error = error.txt
127 # @ initialdir = %s
128 # @ output = output.txt
129 # @ notify_user = smcuser
130 # @ class = long
131 # @ notification = error
132 # @ total_tasks = %i
133 # @ queue
134
135 export PATH=$PATH:/opt/acousto/bin
136
137 acousto -f %s
138
139 """
140
141   open(JOB_SH, \
142       "w").write(script%(os.getcwd(), nprows*npcols, cfg_path))
143
144   #Need for power7 execution. MPI expects an host list file.
145   #One host name per process.
146   f = open("host.list", "w")
147   for n in xrange(nprows*npcols):
148     f.write("power7l\n");
149   f.close()
150
151 def is_first_execution():
152   """
153   Check whether this is the first execution.
154   Must be executed after ./merge.sh
155   """
156   return (len(glob.glob(os.path.join(OUTPUT_DIRECTORY, "*"))) == 0)\
```

```
157           and \
158           (len(glob.glob(os.path.join(BACKUP_DIRECTORY, "*"))) == 0)
159
160 if __name__ == "__main__":
161   if len(sys.argv) != 3:
162     print "Usage:", sys.argv[0], "<template file> <output cfg>"
163     exit(0)
164
165   #compile(template, cfg output)
166   compile(sys.argv[1], sys.argv[2])
```

B.2 fresp.py

```
1  #!/usr/bin/env pypy
2
3  """
4  Take all the single-frequency outputs files and pack
5  the complete frequency response of each mic into
6  his own file.
7
8  Usage: ./fresp.py start_mic end_mic
9
10 Note: The script is _slow_ since for every mic
11       it has to scan through every single frequency
12       file on a line-by-line basis. The slowness is
13       due to high filesystem overhead.
14 """
15
16 import sys, os
17
18 first_mic = 0
19 last_mic = 0
20
21 if len(sys.argv) == 1:
22   print "Usage:", sys.argv[0], "start_mic end_mic"
23   exit(0)
24 elif len(sys.argv) == 2:
25   first_mic = last_mic = int(sys.argv[1])
26 elif len(sys.argv) == 3:
27   first_mic = int(sys.argv[1])
28   last_mic = int(sys.argv[2])
29
30 import glob, math
```

```
31
32 def freqFromName(name):
33   """
34   Get frequency value from file name.
35   """
36   ret = name[name.rfind("-")+1:name.rfind("Hz.out")]
37   return float(ret)
38
39 #We process one mic at a time
40 for mic_index in xrange(first_mic, last_mic+1):
41   #Output file name
42   out_file = "mic%i.out"%mic_index
43   out = open(out_file, "w")
44   freq_counter = 0
45
46   #Iterate through all mics output files
47   for fname in sorted(glob.glob("*mics*.out"), key=freqFromName):
48     with open(fname, "r") as f:
49       while True:
50         line = f.readline()
51
52         #If the current line is the line of the current mic
53         if line.split(" ")[0] == str(mic_index):
54           s = line.split()
55
56           freq = freqFromName(fname)
57           omega = 2 * math.pi * freq
58           re = s[10]
59           im = s[11]
60           tot = s[12]
61
62           newline = "%i %f %f %s %s %s\n"%(freq_counter,
63                                            omega,
64                                            freq,
65                                            re,
66                                            im,
67                                            tot)
68           out.write(newline)
69
70           freq_counter += 1
71           break
72
73   out.close()
```

B.3 max_freq.py

```
"""
Calcola la massima frequenza a cui avere dei buoni risultati
usando la tecnica BEM secondo il criterio mostrato da Katz.

Utilizzo:
Aprire Blender da terminale (altrimenti l'output dello script
non    visibile).
Entrare in 'edit mode' e selezionare l'area di mesh
con cui si vuole lavorare. Tornare in 'object mode'
(Importante! Altrimenti Blender continua a fornire allo script
la selezione precedente. Nel caso fosse una selezione nulla lo
script fallisce con divisione per 0.).
"""

import bpy

sound_speed = 343.0

obj = bpy.context.active_object
edges = obj.data.edges
vertices = obj.data.vertices
matrix = obj.matrix_world.copy()

#Keep only selected edges
edges = [ee for ee in edges if ee.select == True]

#Loop through the edges
mm = 0.0
for ee in edges:
    vv = ee.vertices
    #Transform model coordinates into world coordinates
    v1 = matrix * vertices[ee.vertices[0]].co
    v2 = matrix * vertices[ee.vertices[1]].co
    mm = max((v1-v2).length, mm)

max_freq = sound_speed / (6 * mm)

print("Longest edge's length: %f"%mm)
print("fmax: %i"%int(round(max_freq)))

```

```
41 #Upper bound
42 max_freq = sound_speed / (4 * mm)
43 print("fmax with 4 elements: %i"%int(round(max_freq)))
```

B.4 mirror.py

```
 1 #!/usr/bin/env pypy
 2 # -*- coding: utf-8 -*-
 3
 4 """
 5 Prende un file .nodes di acousto, ne specchia il
 6 contenuto (rispetto a zy) e lo salva in un nuovo file.
 7 """
 8
 9 import re, sys
10
11 if len(sys.argv) != 2:
12   print "Usage:", sys.argv[0], "<nodes file>"
13   exit(0)
14
15 def count_zeros(x):
16   """
17   sum(n < t for n in zero_verts)
18   but somewhat faster
19   """
20   res = 0
21   for z in zero_verts:
22     if z > x:
23       break
24
25     res += 1
26
27   return res
28
29 def get_nodes_info(fname):
30   """
31   Return face/vertex count from file name.
32   """
33   #Format: prefix.<nElem>.<nVert>.nodes
34   s = fname.split(".")
35
36   m = re.search('([a-zA-Z0-9\.\_]+)\.(\d+)\.(\d+)\.(nodes)', fname)
37   return m.group(1), int(m.group(2)), int(m.group(3)), m.group(4)
```

```
38
39 in_name = sys.argv[1]
40 in_prefix, old_nelem, old_nvert, suffix = get_nodes_info(in_name)
41
42 print "Working on file", in_name
43
44 #original vertices and faces (elements)
45 oldv = []
46 olde = []
47
48 #newly generated vertices and faces
49 v = []
50 e = []
51
52 #coincident vertices
53 zero_verts = []
54
55 #vertices processing
56 lasti = 0
57 lines = open(in_name, "r").readlines()
58 for i,line in enumerate(lines):
59   tokens = [float(x) for x in line.split()]
60
61   if len(tokens) != 3:
62     lasti = i
63     break
64
65   oldv.append(tokens)
66   #if the vertex doesn't lie in the
67   #symmetry plane (0, y, z) mirror it
68   #and add it to the new vertices list
69   if tokens[0] != 0:
70     t = tokens[:]
71     t[0] *= -1
72     v.append(t)
73   #if it does lie in the plane, add it
74   #to the list of vertices that need
75   #to be deleted
76   else:
77     zero_verts.append(i)
78
79 #faces processing
80 for line in lines[lasti:]:
```

```
81    tokens = [int(x) for x in line.split()]
82
83    if len(tokens) != 4 and line != "":
84      print "Errore"
85      exit(0)
86
87    olde.append(tokens)
88
89    #Remove concident vertices
90    #and update the others' indexes
91    newtokens = []
92    for t in tokens:
93      if oldv[t-1][0] == 0:
94        newtokens.append(t)
95      else:
96        newtokens.append(t+old_nvert-count_zeros(t-1))
97
98    #Reverse new vertices indexes' order to keep
99    #the normal vector pointing outward
100   newtokens.reverse()
101   e.append(newtokens)
102
103 outf = open(".".join([in_prefix+"_complete",
104             str(len(olde)+len(e)),
105             str(len(oldv)+len(v)),
106             "nodes"]), "w")
107 for vert in oldv:
108   outf.write("%f %f %f\n"%tuple(vert))
109 for vert in v:
110   outf.write("%f %f %f\n"%tuple(vert))
111 for elem in olde:
112   outf.write("%i %i %i %i\n"%tuple(elem))
113 for elem in e:
114   outf.write("%i %i %i %i\n"%tuple(elem))
```

Bibliografia

[1] *G.R.A.S. Sound & Vibration*. http://kemar.us/.

[2] *NextEngine 3D Scanner HD TechSpecs*. http://www.nextengine.com/assets/pdf/scanner-techspecs.pdf.

[3] *ScaLAPACK - Scalable Linear Algebra PACKage*. http://netlib.org/scalapack/slug/index.html.

[4] *IBM Power 770 Server Datasheet*, 2012. http://public.dhe.ibm.com/common/ssi/ecm/en/pod03031usen/POD03031USEN.PDF.

[5] Brian F. G. Katz. Boundary element method calculation of individual head-related transfer function. i. rigid model calculation. 2001.

[6] Brian F. G. Katz. Boundary element method calculation of individual head-related transfer function. ii. impedence effects and comparisons to real measurements. 2001.

[7] Paolo Emilio Mazzon. *The AACSE IBM Power 7*. http://www.dei.unipd.it/~finotell/docs/Il_P770_del_DEI.pdf.

[8] Tianshu Qu, Zheng Xiao, Mei Gong, Ying Huang, Xiaodong Li, and Xihong Wu. Distance-dependent head-related transfer functions measured with high spatial resolution using a spark gap. 2009.

[9] Vincenzo Marchese Umberto Iemma. *AcouSTO (Acoustics Simulation TOol) - User Manual*. http://acousto.sourceforge.net/user_manual/html/UserManual.html.