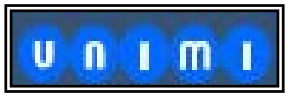




UNIVERSITA' DEGLI STUDI DI MILANO

FACOLTA' DI SCIENZE MATEMATICHE, FISICHE E NATURALI

Corso di Laurea Triennale in
SCIENZE E TECNOLOGIE PER LA COMUNICAZIONE MUSICALE

AUTOCOMPOSIZIONE DI PATTERN RITMICI

Uno strumento informatico per la composizione automatica di ritmi

Relatori:
Dott. Luca Andrea Ludovico

Correlatore:
Dott. Adriano Baratè

Tesi di Laurea di MARCO AICARDI
Anno Accademico 2009-2010

INDICE

1

INTRODUZIONE

Un pattern è generalmente un insieme di oggetti simili disposti in modo da formare una struttura con un grado di regolarità elevato. Può essere pensato come uno schema, un modello o un disegno.

Il concetto è immediatamente rappresentabile nell'ambito spaziale: è un pattern ogni disegno complesso composto con poche forme semplici di cui si variano dimensioni e orientamento, o qualsiasi superficie che presenta caratteristiche uniformi (la corteccia di un albero, la pelliccia di un leopardo, il pavimento di una stanza), o ancora qualsiasi struttura che si rivela costituita da poche sotto-strutture semplici (la disposizione degli occhi di un ragno, le sequenze di DNA o di proteine).

Fig. 1.1: Alcuni pattern spaziali

Schematismi ripetuti sono tuttavia riscontrabili anche nell'ambito temporale: ogni successione di eventi costituita da ripetizioni e/o combinazioni di sequenze di eventi più piccole costituisce un pattern temporale. In questo senso risulta evidente come le arti del tempo (la

musica, la danza, la drammaturgia, la scrittura) siano costruite su pattern: le composizioni musicali, coreografiche, drammatiche, letterarie sono riconducibili a strutture precise che derivano dalla percezione estetica dell'essere umano.

Nella musica in particolare sono rappresentabili come pattern le strutture armoniche e le sequenze melodiche che definiscono i generi, le strutture dei singoli brani, le sequenza ritmiche. Gli schematismi delle strutture armoniche, melodiche, ritmiche e compositive formano l'ossatura dei generi musicali e riflettono la percezione estetica e tratti del grado di evoluzione sociale e culturale di una determinata epoca.

Nel presente lavoro ci si è focalizzati sull'analisi dei pattern nell'ambito ritmico. La disposizione dei colpi di un ritmo all'interno di un intervallo temporale deriva dalle pratiche musicali evolute nei secoli ma segue regole di distribuzione matematiche precise.

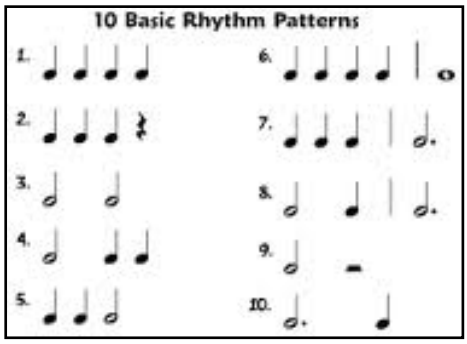


Fig. 1.2: Esempi di pattern ritmici

Dallo studio matematico dei pattern ritmici fondanti della musica occidentale si evidenzia chiaramente come il grado di piacevolezza percepito dipenda dallo schema di distribuzione dei colpi. Gli schemi con maggiore successo, quelli più popolari, risultano essere quelli che contengono un giusto grado di equilibrio tra regolarità e variabilità: ritmi banali annoiano l'ascoltatore, ritmi troppo complessi non gli permettono di essere coinvolto. La caratteristica vincente sembra essere quindi quella di una "variabilità ben strutturata" in grado di coniugare bene le caratteristiche primarie della musica, tensione e distensione, che nel ritmo si manifestano principalmente come differenza ed equivalenza degli intervalli temporali tra i colpi.

L'analisi matematica e geometrica dei pattern ritmici mette in evidenza strutture che sono il punto di partenza per la classificazione dei ritmi e la creazione di algoritmi di riconoscimento automatico. Inoltre vengono ricavate le regole alla base della generazione di pattern ritmici,

da cui è possibile sviluppare algoritmi di composizione automatica e insegnare al calcolatore a comporre ritmi autonomamente, spingendosi nel campo della computer generated music e dell'intelligenza artificiale.

Il presente lavoro mostra alcune importanti tecniche di analisi di pattern ritmici elaborate a partire dalla fine del ventesimo secolo e descrive le caratteristiche di un programma informatico (scritto in codice Objective-C) che svolge il compito di creare automaticamente pattern ritmici a partire da pochi semplici parametri forniti dall'utente.

2

ANALISI di PATTERN RITMICI

2.1 Cos'è il ritmo

Lat.: rhythmus, che è il greco rhythmòs: "ordine nel movimento", "proporzione regolare", "giusta misura".
Ritmo: "lo scandirsi, il succedersi di un fenomeno a intervalli regolari di tempo" [1];
"... movimento caratterizzato da una successione regolare di elementi forti ed elementi deboli o di opposte o differenti condizioni."[2].
Ogni successione di stati differenti dà vita ad un ritmo e di conseguenza il termine ha un'ampiezza semantica pressochè sterminata. Specularmente al concetto di 'pattern', ha a che fare principalmente col tempo, ma anche con lo spazio, è usato in tutte le arti, soprattutto in musica e in danza, ma anche in poesia (il ritmo dei versi), nelle arti figurative (l'alternarsi delle tonalità e dei volumi nella pittura e nella scultura), in architettura, nel cinema, nel romanzo (il ritmo dell'intreccio); si ritrova anche in contesti diversi da quelli artistici, come in astronomia (si parla di ritmi delle stagioni). Esistono innumerevoli riferimenti al ritmo anche nella tradizione filosofica e matematica. Il ritmo fà parte della nostra vita di tutti i giorni, sia esteriormente (il ritmo del giorno e della notte), che interiormente (il ritmo del cuore, quello della respirazione).
Nell'ambito di questo lavoro non prenderemo in considerazione tutte queste possibili accezioni, ma ci riferemo esclusivamente all'ambito musicale dove la sua definizione è più ridotta. La parola "ritmo" in musica è un termine che si riferisce ad una successione di eventi musicali.

Solo la successione di più fenomeni sonori può possedere un ritmo in quanto è evidente che un suono istantaneo non ha ritmo; la parola indica dunque il rapporto che esiste tra i diversi elementi in una sequenza di eventi musicali successivi.

Breve storia del ritmo [3]

La scansione delle musica dell'antichità greco-romana si plasmava in buona parte sul ritmo poetico, risultante nell'organizzazione delle differenti durate temporali delle singole sillabe (distinte in brevi e lunghe). Questo sistema ritmico viene normalmente definito come sistema del ritmo libero.

Questa concezione del ritmo fu completamente superata con la progressiva perdita della percezione della durata sillabica e con la trasformazione dell'accento da melodico in percussivo.

Nel Medioevo prese piede un sistema ritmicamente misurato basato sulla distinzione qualitativa tra accenti ritmici forti e accenti ritmici deboli. In realtà, sebbene non fosse stato esplicitamente teorizzato, il ritmo caratterizzato dall'alternarsi regolare degli accenti forti e deboli o di gruppi di tali accenti, doveva essere operante da sempre nelle marce e nelle danze di carattere popolare.

Una completa trasformazione si ebbe con l'introduzione dei modi ritmici e della notazione mensurale (secc. XII-XIII) che configurarono un ritmo basato su valori temporali multipli o sottomultipli di una data unità di tempo, organizzato in schemi di progressiva complessità e sulla base di una grande indipendenza ritmica tra le voci del'ambito polifonico.

Con il passaggio dalla scrittura contrappuntistica alla melodia accompagnata, tra la fine del '500 e l'inizio del '600, il ritmo venne radicalmente semplificato attraverso l'introduzione della battuta e l'elaborazione di una concezione ritmica unitaria, basata sulla regolare successione di unità metriche uniformi, ciascuna caratterizzata da una costante collocazione degli accenti. A questa concezione del ritmo si è attenuta a partire dal secolo XVII la musica moderna occidentale, che ha tuttavia progressivamente temperato la rigidezza dei suoi schemi di base fino a riconquistare, nel secondo '800, una dimensione ritmica più versatile e complessa che era stata tipica della tradizione contrappuntistica nei secoli XIV - XVI.

2.2 L'analisi del ritmo musicale

L'analisi matematica e geometrica del ritmo aiuta a mettere in luce alcune sue caratteristiche fondamentali e fornisce tecniche utili al riconoscimento automatico e alla generazione di pattern ritmici.
I ritmi che userò come esempi sono alcuni dei ritmi fondamentali che stanno alla base delle strutture dei ritmi contemporanei. Essi provengono principalmente dalla tradizione musicale africana, per via diretta o rielaborata nella tradizione sudamericana, e sono composti solitamente con un unico strumento. Mentre nella cultura africana lo strumento utilizzato è un campanaccio di ferro, in quella sudamericana vengono usate due bacchette di legno duro fatte sbattere una contro l'altra, chiamate Clave. Dei diversi ritmi composti con le Clave uno in particolare ha avuto particolare diffusione, il Clave Son di Cuba. Negli esempi di rappresentazione che seguono vedremo principalmente questo ritmo.

2.2.1 Rappresentazione del ritmo

Esistono diverse possibili rappresentazioni di un pattern ritmico.
Nella notazione occidentale si mutua la notazione ritmica da quella di tipo melodico-armonico, utilizzando come strumento grafico di rappresentazione il pentagramma. I colpi degli strumenti che contengono una caratteristica di intonazione musicale (i tamburi) sono rappresentati come note di diversa altezza, rappresentando gli strumenti con tonalità più grave come note più gravi e viceversa. Per quanto riguarda invece gli strumenti che non presentano una caratteristica tonale (i piatti), il corpo della nota viene simboleggiato da una "x" anzichè dal classico pallino di colore nero o bianco. La durata del colpo è rappresentata utilizzando la stessa simbologia della rappresentazione melodica.

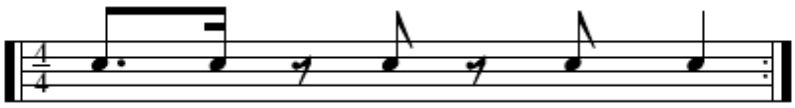


Fig. 2.1: un esempio di notazione ritmica occidentale.

Al posto del pentagramma, sempre nella notazione occidentale, si usa talvolta una sua derivazione semplificata costituita da una sola riga al posto di cinque, in cui ogni strumento della batteria occupa un proprio "monogramma". Questo tipo di rappresentazione ci avvicina allo studio dell'essenza del ritmo, in quanto risulta più chiara nel mostrare le caratteristiche sequenziali del ritmo e si presta meglio all'analisi dei ritmi più semplici, quelli primordiali da cui poi si sono sviluppati i ritmi complessi di oggi.

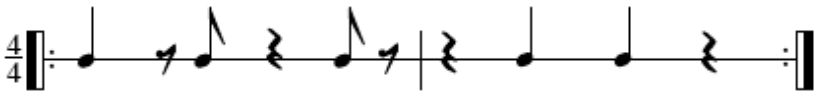


Fig. 2.2: un secondo esempio di notazione ritmica occidentale.

Per lo studio del ritmo sono però più funzionali altri tipi di rappresentazione, non facenti parte della notazione musicale occidentale [4] [5].
Una è la Box Notation Method, sviluppata da Philip Harland presso la University of California di Los Angeles nel 1962, conosciuta anche come TUBS (Time Unit Box System). Questo tipo di rappresentazione è utilizzata molto dagli etnomusicologi [5] ed è molto utile ai percussionisti che non hanno familiarità con la notazione occidentale.

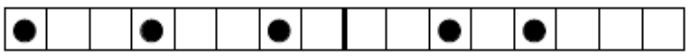


Fig. 2.3: Box Notation Method.

Questo tipo di notazione, in una sua comune variante in cui i colpi sono rappresentati con un simbolo e le pause con un altro [6], viene utilizzata negli esperimenti di psicoacustica legati alla percezione del ritmo,

X . . X . . X . . . X . X . . .

Fig. 2.4: variante della Box Notation Method.

Un' altra variante che ci rimanda immediatamente al campo dell'informatica è quella binaria in cui un ritmo composto da sedici battiti è rappresentato come sequenza di 16 bit in cui i colpi sono rappresentati da 1 e le pause da 0.

1 0 0 1 0 0 1 0 0 0 1 0 1 0 0 0

Fig. 2.5: variante binaria della Box Notation Method.

Un' ulteriore rappresentazione si ricava focalizzando l'attenzione, anzichè sulla presenza o assenza di colpi, sulla durata degli intervalli tra un colpo e l'altro.
I numeri rappresentano la lunghezza degli intervalli tra i colpi. Questa rappresentazione numerica degli intervalli ha grandi vantaggi dal punto di vista della compattezza ma è abbastanza povera dal punto di vista grafico.

3 3 4 2 4

Fig. 2.5: rappresentazione numerica degli intervalli.

Rappresentazione geometrica

Dato che la notazione musicale tradizionale occidentale non si presta a rappresentare visivamente il ritmo, sono state create forme grafiche alternative che non solo permettono di manipolare e visualizzare meglio il ritmo, ma si prestano anche ad essere utilizzate al di fuori del dominio prettamente musicale. Queste rappresentazioni hanno origine o derivano da culture ben precedenti a quelle che hanno inventato la notazione occidentale. Ad esempio in un libro arabo del 1252 scritto da Safi-al-Din il ritmo è rappresentato su un cerchio diviso in settori circolari [7]. Sotto vediamo il ritmo di Clave Son rappresentato in questa notazione, dove i settori che contengono un colpo sono evidenziati in colore grigio. La freccia rappresenta l'inizio del ritmo e la direzione di lettura.

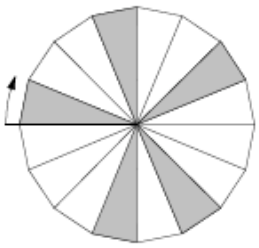


Fig. 2.6: rappresentazione geometrica araba del clave Son (1252).

Nel 1987 Kjell Gustafson del laboratorio di Fonetica della Oxford University, pubblicò un metodo originale di rappresentare il ritmo su un grafico bidimensionale [8] [9]. Utilizziamo come esempio il ritmo clave Son. Consideriamo inizialmente la rappresentazione testuale della sequenza degli intervalli vista precedentemente. Le lunghezze degli intervalli tra un colpo e il successivo (3, 3, 4, 2, 4) sono ordinate secondo l'asse delle x. Questa rappresentazione mostra bene in quale sequenza avvengono i colpi, ma non risulta molto chiara la durata dei colpi stessi. Utilizzando quindi un grafico ad istogramma le lunghezze degli intervalli saranno rappresentate sull'asse y, risultando in una rappresentazione denominata Spettro degli Intervalli Adiacenti.

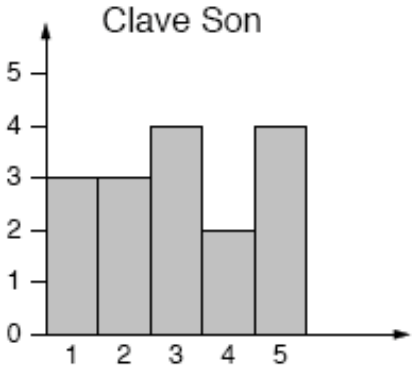


Fig. 2.7: spettro degli intervalli adiacenti.

In questo grafico le lunghezze relative degli intervalli sono chiaramente visibili, ma viene persa l'informazione temporale lungo l'asse x. Per avere una rappresentazione che avesse i vantaggi di entrambe Gustafson decise di combinarle.

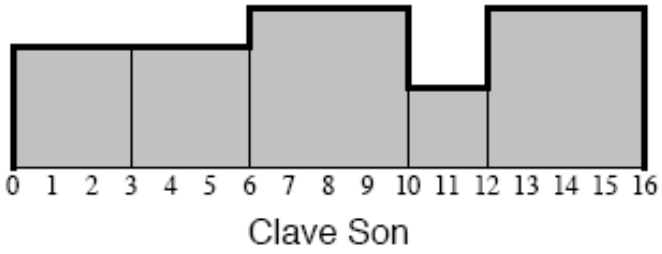


Fig. 2.8: rappresentazione TEDAS.

Ogni intervallo è rappresentato da un cubo e le durate temporali sono chiaramente visibili sia sull'asse x che sull'asse y. Gustafson chiamò questa rappresentazione TEDAS (Temporal Elements Displayed As Squares). Questo tipo di grafico oltre ad avere una buona immediatezza visiva presenta anche altri vantaggi, tra cui alcuni legati alla misura di somiglianza tra ritmi differenti.

Nel 2002 Hofmann-Engl, che non erano al corrente del lavoro di Gustafson, riscoprì l'idea di rappresentare un ritmo come grafico bidimensionale dove entrambi gli assi misurano il tempo. Mentre il lavoro di Gustafson rispondeva ad esigenze prettamente di visualizzazione, Hofmann-Engl si occupava della predizione della percezione e riconoscimento dell'uomo della somiglianza tra i ritmi. Pur con due approcci diversi i risultati ottenuti in termini di rappresentazione sono gli stessi.

Fig. 2.9: rappresentazione a catena cronotonica.

Il ritmo viene innanzitutto frammentato in unità base (in questo caso 16 unità, tra colpi e pause), chiamate "atomic beats" (battiti atomici). Ad ogni atomic beat viene associato un valore y pari alla lunghezza dell'intervallo di cui fà parte, andando così a comporre un pattern di sedici punti sul piano. Infine questi punti sono collegati ognuno al successivo con l'incrementare del valore

di x in modo da ottenere una catena poligonale. Hofmann-Engl definisce queste curve poligonali come “catene cronotoniche”.

Un terzo modo di rappresentare geometricamente un ritmo è quello del “poligono convesso”. Come la rappresentazione cronotonica di Gustafson e Hofmann-Engl, quella poligonale mostra bene la lunghezza relativa degli intervalli tra un colpo e il successivo e il loro succedersi sulla scala temporale circolare. Questa rappresentazione risulta ancora più appropriata delle sue corrispondenti versioni cronotoniche in quanto aggiunge la visualizzazione delle ciclicità del ritmo, ponendo come adiacenti l’inizio e la fine del ritmo, mentre nelle rappresentazioni precedenti inizio e fine erano i punti più distanti.

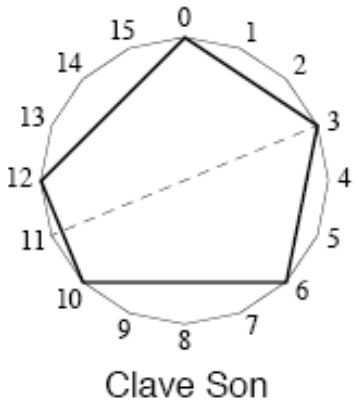


Fig. 2.10: rappresentazione a poligono convesso.

Questa rappresentazione è uno strumento molto potente di visualizzazione in quanto osservando i poligoni diverse proprietà geometriche del ritmo risultano immediatamente percepite.

Ad esempio la linea tratteggiata tra i punti 3 e 11 in figura mostra che il ritmo clave Son contiene un asse di simmetria. Dal poligono si possono anche ottenere svariati vettori di proprietà geometriche $X = (x_1, x_2, ..., x_n)$. Ad esempio, ponendo l’attenzione sui vertici del poligono, si potrebbe costruire il vettore $V = (v_1, v_2, ..., v_n)$ dove v_i è l’angolo interno dell’i-esimo vertice. Oppure si potrebbero considerare i lati e costruire un vettore $L = (l_1, l_2, ..., l_n)$ dove l_i è la lunghezza dell’i-esimo lato. Inoltre si potrebbero usare proprietà globali della forma geometrica come il rapporto tra il perimetro e l’area o il momento di inerzia. Proprietà come quelle appena viste sono di recente state utilizzate per una nuova classificazione delle sequenze ritmiche appartenenti alla musica tradizionale africana e afro-americana [11] [12].

2.2.2 La misura dell'uniformità ritmica

Proviamo ora a considerare tre ritmi in dodici ottavi, espressi in notazione "box":

[x . x . x . x . x . x .]

[x . x . x x . x . x . x]

[x . . . x x . . x x x .]

Risulta intuitivamente chiaro che il primo ritmo è più uniformemente distribuito del secondo e il secondo è più uniformemente distribuito del primo. Il secondo è derivato dalla cultura africana ed è uno dei ritmi più popolari al mondo. Il suo nome, derivato dalla sua versione Cubana, è Bembè [12]. In genere i ritmi della tradizione hanno la proprietà di avere un elevato grado di distribuzione uniforme. Per questo motivo la misura matematica dell'uniformità di distribuzione, insieme ad altre proprietà geometriche, viene utilizzata nel campo della etnomusicologia matematica [13] [14] per identificare, se non addirittura spiegare, le preferenze culturali per alcuni ritmi della musica tradizionale.

Ritmi a massima distribuzione

Il lavoro sulla misura della distribuzione di un ritmo deriva alcuni suoi principi dalla ricerca svolta sulla uniformità di distribuzione dell'altezza delle note [15] all'interno di una scala musicale. Duthett e Entringer hanno costruito diverse misure matematiche della quantità di distribuzione all'interno di una scala musicale. Una delle loro misure somma semplicemente la lunghezza di tutti gli archi di cerchio ottenuti da tutte le coppie di valori di altezza all'interno della scala musicale. Questo tipo di misura può facilmente essere riportato nell'ambito delle durate degli intervalli di ritmi ciclici rappresentati su una circonferenza. Tuttavia questo tipo di misura è troppo grezzo [11] [12] per rappresentare correttamente i ritmi della tradizione africana e sudamericana. La misura è in grado di fornire valori differenti tra ritmi che già si differenziano notevolmente l'uno dall'altro. Ad esempio i due ritmi costituiti da 4 colpi :

[x . . . x . . . x . . . x . . .] e [x . x . x . . x . . . . . .] producono valori di distribuzione pari a 32 e 23 rispettivamente, mostrando chiaramente che il primo ritmo è più distribuito del secondo.

Tuttavia tutti i 6 pattern fondamentali di Clave da quattro quarti illustrati in figura risultano avere un valore di distribuzione pari a 48, ma percepiamo chiaramente il pattern di Rumba come meno distribuito del pattern di Bossa-Nova.

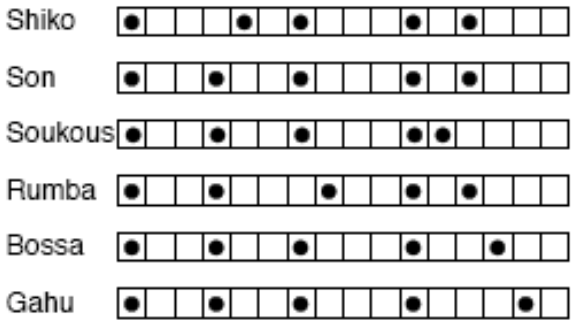


Fig. 2.11: i 6 pattern fondamentali di Clave da 4/4.

Se al posto della misura degli archi di cerchio utilizziamo la misura delle corde (lati e diagonali del poligono), come proposto da Block e Douthet [16], la misura risulta molto più discriminante e viene quindi considerata come la misura migliore di distribuzione ritmica.
Il metodo di misura di Block e Douthet risulta inoltre confermato da un lavoro del matematico ungherese Fejes Toth [17] il quale ha dimostrato che la massimizzazione della somma delle coppie di corde del poligono (quindi la massima distribuzione del ritmo) si ottiene quando gli n colpi sono distribuiti a formare un poligono regolare di n lati.

2.2.3 Gli spettri di intervalli

Piuttosto che sulla somma degli intervalli tra coppie di colpi, concentriamo ora la nostra attenzione sulla forma dello spettro della frequenza con cui un certo intervallo è presente all'interno di un ritmo. Consideriamo sempre i ritmi rappresentati come punti su una circonferenza. Nella teoria musicale questo spettro è chiamato "interval vector" (vettore di intervalli) [18]. Per esempio, l'interval vector del pattern del clave Son è costituito da [0, 1, 2, 2, 0, 3, 2, 0]. Il vettore è 8-dimensionale in quanto ci sono 8 possibili intervalli se consideriamo coppie di colpi definite su una circonferenza di sedici battiti. Nel caso del clave Son abbiamo 5 colpi (corrispondenti a 10 possibili coppie di colpi), quindi la somma degli elementi del vettore è uguale a 10. Una utile visualizzazione dello spettro si ottiene rappresentando il vettore su un istogramma, come in figura.

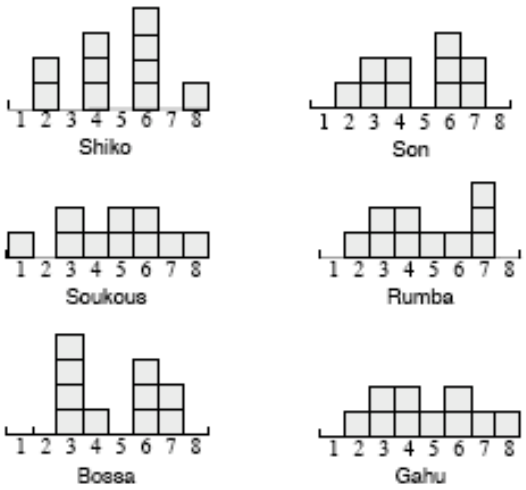


Fig. 2.12: gli istogrammi dei pattern di Clave da 4/4.

L'esame dei sei istogrammi ci porta a porci interessanti domande nei diversi campi di ricerca della musicologia, della geometria,del calcolo combinatorio e della teoria dei numeri. Ad esempio, David Locke [19] ha fornito spiegazioni di tipo musicale alla caratterizzazione del pattern del ritmo Gahu, definendolo "ritmicamente potente", in grado di suscitare un "effetto a spirale", dal "fraseggio ambiguo" che porta ad "illusioni acustiche". Se confrontiamo l'istogramma del pattern Gahu con gli altri sei notiamo che è l'unico che ha un'altezza massima pari a 2 e che è costituito da un blocco unico di celle occupate nell'istogramma. L'unico altro ritmo costituito da un singolo blocco è la Rumba, ma ha 3 intervalli di lunghezza 7. L'unico altro ritmo con un altezza massima dell'istogramma pari a 2 è il Soukous, ma è costituito da 2 blocchi, dato che non ha alcun intervallo di lunghezza 2. Solo i pattern Soukous e Gahu usano 7 intervalli degli 8 disponibili.

Da queste osservazioni possiamo ipotizzare che probabilmente altri ritmi che presentano istogramma piatto e pochi o nessun buco potrebbero essere interessanti dal punto di vista musicologico. Se la forma dell'istogramma spettrale del ritmo di Gahu è legata alle sue caratteristiche musicali, si potrebbe utilizzare questa caratteristica geometrica per identificare o generare automaticamente altri patterns con qualità simili e creare uno strumento per la composizione automatica svolta da un computer. Una domanda che ci potremmo fare è se esistono ritmi che hanno le proprietà viste sopra (istogramma piatto e a blocco unico) al grado massimo possibile. Definiamo la famiglia dei ritmi costituiti da k colpi in un tempo di n battiti

come R[k,n]. Come esempio, tutti i pattern di Clave in quattro quarti visti sopra appartengono alla famiglia ritmica R[5,16].

Una delle prime domande che ci possiamo fare è se esistono dei ritmi con spettro piatto di altezza 1 e costituiti da un unico blocco. Questo evidentemente non è possibile in R[5,16]: dato che ci sono solo 8 possibili lunghezze di intervallo differenti e dieci coppie di distanze tra colpi, ci deve essere almeno una cella dell'istogramma con altezza maggiore di 1. Una seconda domanda che sorge naturalmente è se esiste in R[5,16] un ritmo che utilizza tutte e 8 le possibili lunghezze di intervallo. La risposta in questo caso è affermativa; un pattern con queste caratteristiche è [x x . . . x . x . . . . . x . .], che ha un interval vector uguale a [1, 1, 1, 2, 1, 2, 1, 1]. Cambiando famiglia e prendendo la R[4,12] troviamo che il ritmo [x x . . x . x . . . . .] ha un istogramma con tutte altezze pari a uno. Il suo interval vector è infatti [1, 1, 1, 1, 1, 1].

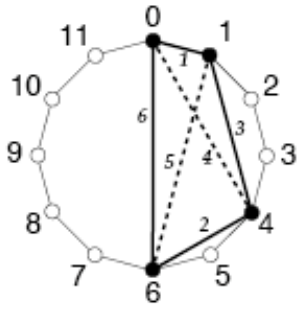


Fig. 2.13: un ritmo ad istogramma piatto.

Un ritmo trascinante non deve ovviamente contenere intervalli di silenzio troppo lunghi, come quello presente alla fine del pattern appena visto. Se proviamo a cercare un pattern di R[4,12] con interval vector [1, 1, 1, 1, 1, 1] che non contenga intervalli di silenzio così lunghi, troviamo il ritmo [x x . x . . . x . . . .] mostrato in figura. L'intervallo di silenzio più lungo è di 5 unità.

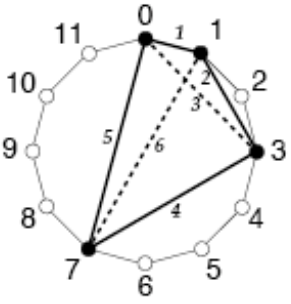


Fig. 2.14: un ritmo ad istogramma piatto, con intervallo di silenzio minimizzato.

Una sequenza ciclica come [x x . . x . x . . . . .] è un esempio di struttura chiamata “collana” [20] con “perle” di 2 colori; ed è anche un esempio di una struttura di tipo “braccialetto”. Due collane sono considerate equivalenti se ruotando una delle due si riescono a sovrapporre perfettamente tutte le perle, in una corrispondenza uno a uno. Due braccialetti sono considerati equivalenti se si riescono a mettere le loro perle in corrispondenza uno a uno ruotandone e/o rivoltandone (immagine speculare) uno dei due.
Se ruotiamo i pattern appena visti, il loro interval vector rimane evidentemente lo stesso, ma i ritmi prodotti suonano in maniera leggermente differente. Per questo è utile distinguere tra collane ritmiche e ritmi fissati in una certa posizione di rotazione (rispetto alla struttura temporale sottostante).
Nel calcolo combinatorio il numero di colpi all’interno di un pattern si definisce Densità ed esitono algoritmi in grado di generare tutte le collane con una specifica densità [21].

Ritmi con specifiche molteplicità intervallari

Scale musicali le cui altezze sono determinate da punti disegnati su una circonferenza e che hanno la proprietà di Erdos (con n numero di punti disegnati sulla circonferenza, ogni distanza intervallare è l’unica ad occorrere un numero i di volte, con i che và da 1 a n-1) viene chiamata Deep Scale [20]. Possiamo portare questo concetto dall’ ambito delle frequenze a quello temporale, definendo come Deep Rhithm un ritmo che possiede la stessa proprietà. Una famosissima Deep Scale è la scala diatonica occidentale, il cui corrispettivo ritmico è il pattern di Bembè (scala diatonica occidentale e Bembè sono isomorfi).

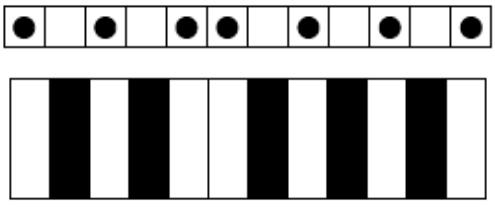


Fig. 2.15: ismorfismo del pattern di Bembè e della scala diatonica.

Due gruppi di punti non congruenti, come le collane viste precedentemente (Fig. 2.13 e 2.14) sono detti omometrici se l'insieme delle molteplicità delle distanze tra i loro punti è identico, ovvero hanno spettri di intervalli identici. Alcuni risultati legati ai pattern omometrici sono di rilievo nella teoria del ritmo. Ad esempio molti pattern ritmici sono costituiti da due suoni (come la conga alta e la conga bassa) che insieme occupano ogni battito all'interno della battuta. E' un risultato noto che ogni insieme di n vertici in un poligono regolare di 2n lati è omometrico al suo complementare [22].
Questo porta ad un semplice algoritmo per generare un pattern composto di due ritmi complementari in cui i due pattern sono omometrici.

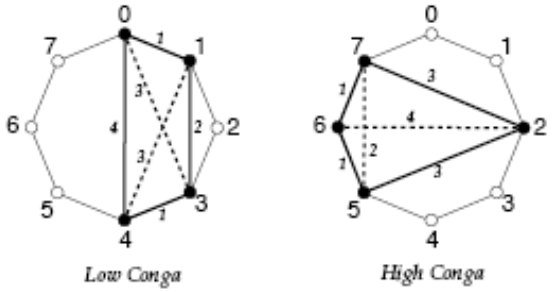


Fig. 2.16: omometricità di un pattern di conga alta e conga bassa.

E' noto anche che 2 ritmi sono omometrici se e solo se anche i loro complementari lo sono [23]. Questo porta ad un altro metodo interessante per generare automaticamente ritmi tramite computer.

2.2.4 Misurare la somiglianza tra i ritmi

Alla base di ogni algoritmo di comparazione, riconoscimento e classificazione di ritmi stà la misura della somiglianza tra coppie di ritmi. Il tipo di misura di somiglianza scelto è in parte determinato dal modo in cui il ritmo è rappresentato.
Esistono una grande varietà di metodi per misurare il grado di somiglianza di due ritmi rappresentatio come stringhe di simboli. Il problema del pattern matching è un problema classico nel campo del riconoscimento dei pattern e della scienza dei computer in generale. Tradizionalmente si misura la somiglianza di due pattern tramite una semplice operazione di confronto con modelli. Più di recente il grado di somiglianza è stato misurato con funzioni

maggiormente complesse e potenti come la “earth mover’s distance” [24] [25].
Descriviamo di seguito le cinque misure più popolari (e semplici da calcolare) del grado di somiglianza tra pattern ritmici.

La distanza di Hamming

Una misura della distanza di similarità spesso usata nella teoria dei codici per sequenze binarie è la distanza di Hamming [26]. La distanza di Hamming è molto semplicemente il numero di punti all’interno della stringa per i quali i valori non corrispondono (si può usare il numero di corrispondenze secondo la convenienza). E’ uno dei modi più semplici di confrontare schemi.
Si considera ogni ritmo rappresentato da un vettore $X = (x_1, x_2, ..., x_n)$ dove x_i rappresenta il valore binario del ritmo in quel punto. Se un colpo viene suonato al tempo i allora $x_i = 1$, altrimenti $x_i = 0$. Un ritmo viene quindi rappresentato come punti di uno spazio vettoriale n-dimensionale a valori binari (chiamato ipercubo). La distanza di Hamming tra due punti $X = (x_1, x_2, ..., x_n)$ e $Y = (y_1, y_2, ..., y_n)$ di questo spazio è data da:

$$d_H(X,Y) = \sum_{i=1}^{n} |x_i - y_i|$$

dove |x| indica il valore assoluto di x.
La distanza di Hamming è veloce da calcolare (in tempo O(n)) ma non è l’ideale per il nostro problema di grado di somiglianza ritmica, quando utilizzata con questa rappresentazione vettoriale, in quanto anche se rileva la presenza di una dissimiglianza non misura la sua distanza.

La distanza Euclidea tra vettori di intervalli

Alcuni algoritmi di identificazione di ritmi [27] e sistemi di riconoscimento da parte del computer di pattern ritmici [28] considerano gli intervalli inter-onset la caratteristica fondamentale per misurare il grado di somiglianza. Gli intervalli inter-onset sono gli intervalli di tempo tra i colpi consecutivi all’interno di un ritmo. Questo approccio e le sue varianti sono più appropriati rispetto alla distanza di Hamming per la misurazione del gradi di somiglianza di due

ritmi. Rappresentiamo un ritmo con un vettore di valori numerici che rappresentano questi intervalli. Un ritmo sarà rappresentato nello specifico da un vettore $X = (x_1, x_2, ..., x_n)$ dove x_i sono il numero di vertici saltati del poligono di n lati per arrivare al colpo successivo. Questi vettori sono sostanzialmente analoghi a quelli che contengono le sequenze di intervalli temporali in quanto questi sono uguali al numero di vertici saltati più uno. Il grado di somiglianza tra due ritmi $X = (x_1, x_2, ..., x_n)$ e $Y= (y_1, y_2, ..., y_n)$ può essere misurato dalla distanza Euclidea tra i due vettori di intervalli X e Y, in questo spazio vettoriale n-dimensionale, dato da:

$$d_E(X,Y) = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}$$

I vettori di intervalli a partire dalle sequenze binarie sono calcolati in tempo O(n), dopodichè la distanza Euclidea può essere calcolata in un tempo O(k), dove k è il numero di colpi. Il tempo complessivo di calcolo sarà quindi O(n).

La distanza di vettori di differenze intervallari

Nel lavoro di Coyle e Shmulevic [28], e di Shmulevic et al. [29] un pattern musicale è rappresentato da un vettore da loro chiamato "vettore di differenze ritmiche". Se $T = (t_1, t_2, ..., t_n)$ è un vettore di intervalli temporali tra coppie di colpi appartenenti ad un ritmo, allora viene definito il vettore $X = (x_1, x_2, ..., x_n)$ in cui $x_i = t_i+1/t_i$. Data la cilcicità dei ritmi che stiamo prendendo in considerazione il valore di x_n corrisponde a t_n/t_1, e quindi la dimensione del vettore di differenze ritmiche è n, non n-1. Utilizzeremo d'ora in poi il più breve e maggiormente descrittivo termine "vettore di differenza intervallare" al posto di "vettore di differenze ritmiche". Utilizziamo i vettori $X = (x_1, x_2, ..., x_n)$ e $Y = (y_1, y_2, ..., y_n)$ per definire due vettori di differenza intervallare. Shmulevic et al. hanno definito il grado di somiglianza tra X e Y come:

$$d_{ID}(X,Y) = \left(\sum_{i=1}^{n}\frac{max(x_i,y_i)}{min(x_i,y_i)}\right) - n$$

Anche questa distanza è facilmente calcolata in tempo O(n).

La "swap distance"

Per fare fronte alla intrinseca debolezza della distanza di Hamming, sono state proposte negli anni molte sue varianti e generalizzazioni. Una delle prime generalizzazioni è stata la distanza di edit (edit distance), che ammette di inserire e cancellare colpi all'interno del ritmo [30] [31]. Una generalizzazione più recente e degna di nota è la distanza di Hamming fuzzy (fuzzy Hamming distance) [32] [33], la quale permette, oltre alle inserzioni e cancellazioni dei colpi, di spostarne la posizione. Tali modi di calcolare la distanza si possono calcolare in tempo $O(n^2)$.
Questo modo completamente diverso di misurare il grado di somiglianza di due stringhe calcola la quantità di "lavoro" necessaria a trasformare una stringa nell'altra. Questo tipo di approccio è comune nella bioinformatica dove le due stringhe da comparare sono catene di polimeri e il lavoro misura il minimo numero di operazioni di base richieste per trasformare una molecola in un' altra. Il tipo di operazione di base presa in considerazione varia e di solito modella alcuni tipi di mutazione legati all'evoluzione [34] [35]. Qui utilizzeremo una versione semplificata della distanza di Hamming fuzzy che è molto più efficiente calcolare.
Il problema di comparare 2 stringhe binarie della stessa lunghezza e con lo stesso numero di uno suggerisce l'utilizzo di un tipo di operazione molto semplice, chiamato "swap". Uno swap è uno scambio di posto tra un uno e uno zero che hanno posizioni adiacenti nella stringa binaria.
Scambiare la posizione di elementi di stringhe numeriche è un'operazione fondamentale in molti algoritmi di ordinamento [36]. Tuttavia, nella letteratura legata all'ordinamento, uno swap può scambiare elementi non adiacenti. Quando è richiesto che gli elementi da scambiare siano adiacenti, lo swap viene chiamato mini-swap o swap primitivo [37] [38]. Qui utilizzeremo il termine più breve swap per indicare lo scambio di due elementi adiacenti. La swap distance tra due ritmi è il numero minimo di swap richiesti per trasformare un ritmo nell'altro. Ad esempio il ritmo [x . x . x x . x . x . x] può essere trasformato nel ritmo [x . x x . x x . x . x .] con un numero minimo di swap uguale a 4, interscambiando il terzo, il quinto, il sesto e il settimo colpo con le corrispondenti pause che li precedono. La swap distance può essere vista come una distanza di Hamming fuzzy semplificata [32] [33], dove sono svolte solo le operazioni di spostamente e il costo dello spostamento è pari alla sua lunghezza. Una misura del grado di somiglianza siffatta sembra essere più appropriata della distanza di Hamming tra vettori binari o la distanza Euclidea

tra vettori intervallari nel contesto della somiglianza ritmica [11] [12]. La swap distance può anche essere considerata come un caso speciale della più generale Earth Mover's Distance (distanza di movimento terra, anche chiamata "distanza di trasporto") usata da Typke et al. per misurare la somiglianza melodica [25]. Dati due gruppi di punti chiamati "punti di rifornimento" e "punti di richiesta", con assegnati date quantità di peso di materiale, la distanza di movimento terra misura la minima quantità di lavoro richiesta per trasportare materiale dao punti di rifornimento ai punti di richiesta. Nessun punto di rifornimento può fornire più peso di quanto possieda e nessun punto di domanda può ricevere più peso di quanto abbia bisogno. La swap distance è la versione unidimensionale della distanza di movimento terra , con tutti i pesi pari a uno. Inoltre, nel caso in cui entrambe le sequenze binarie abbiano lo stesso numero di uno (o di colpi) esiste una corrispondenza uno a uno tra gli indici dei colpi ordinati delle sequenze. Per esempio consideriamo ancora le due sequenze X = [x . x . x x . x . x . x] e Y = [x . x x . x x . x . x .], ognuna contenente sette colpi. L'i-esimo colpo di X deve percorrere una certa distanza per raggiungere la posizione dell'i-esimo colpo di Y. Per i = 1 questa distanza è nulla, per i = 3 la distanza è 1, etc... .

La swap distance può essere calcolata svolgendo realmente gli scambi, ma è un processo poco efficiente. Se X ha degli uno nella prima metà della sequenza e zero altrove, e se Y ha degli uno nella seconda parte della sequenza e zero altrove, allora almeno un numero quadratico di scambi sarà richiesto. D'altro canto, se invece noi compariamo le distanze, scopriamo un algoritmo decisamente più efficiente. Per prima cosa percorriamo la sequenza binaria e salviamo un vettore che contiene le coordinate x a cui i k colpi avvengono. Per esempio, gli X e Y di cui sopra generano i vettori U = (u_1, u_2, ..., u_7) = (1, 3, 5, 6, 8, 10, 12) e V = (v_1, v_2, ..., v_7) = (1, 3, 4, 6, 7, 9, 11) rispettivamente. La differenza tra ui e vi è il numero di scambi che devono essere svolti per allineare i colpi. Quindi, in generale, la swap distance tra due vettori di coordinate di colpi U e V con k colpi è data da:

$$d_{SWAP}(U, V) = \sum_{i=1}^{k} |u_i - v_i|$$

Il calcolo di U e V a partire da X e Y si svolge facilmente in tempo O(n) con un semplice passaggio. Quindi la distanza di swap si riesce a calcolare in tempo O(n), con un notevole risparmio di calcolo.

La "chronotonic distance"

Hofmann-Engl considera la catena cronotonica come un vettore costituito da unità atomiche, piuttosto che una funzione lineare composta di varie parti. Per esempio, il ritmo di clave Son che ha il vettore di intervalli (3,3,4,2,4) e la corrispondente catena cronotonica in figura, genera il vettore cronotonico 16-dimensionale (3,3,3,3,3,3,4,4,4,4,2,2,4,4,4,4).

Fig. 2.17: catena cronotonica del clave Son.

Per misurare il grado di somiglianza tra due ritmi Hofmann-Engl calcola la distanza Euclidea pesata tra i corrispondenti vettori cronotonici. Hofmann-Engl ci riporta che esperimenti condotti su soggetti umani supportano l'ipotesi che questa misura combacia con i modelli cognitivi e percettivi di somiglianza ritmica umani [10]. Tuttavia, osservare la rappresentazione cronotonica in forma di funzione, come la funzione composta da diverse parti in figura 2.18, apre le porte ad una vasta famiglia di possibili funzioni di calcolo della distanza con già utilizzate nei campi della statistica e del riconoscimento dei pattern.

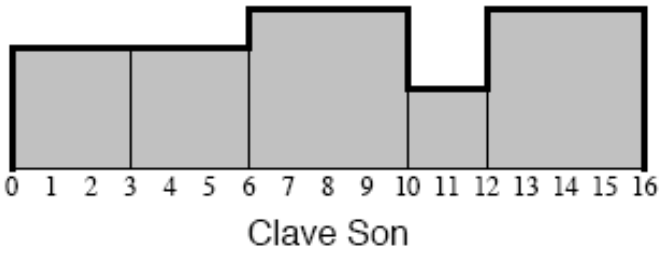


Fig. 2.18: rappresentazione TEDAS del clave Son.

Date due funzioni di densità probabilistica f1(x) e f2(x), esistono diverse misure della distanza tra l'una e l'altra. Una delle più conosciute è la "informazione di discriminazione". L'informazione di discriminazione tra f1(x) e f2(x), anche detta "divergenza di Kullback-Liebler" tra le distribuzioni [39], è data da:

$$KL = \int f_1(x) \log \frac{f_1(x)}{f_2(x)} dx.$$

Questa misura è strettamente relazionata alla distanza variazionale di Kolmogorov, data da:

$$K = \int |f_1(x) - f_2(x)| dx.$$

E' possibile anche calcolare la distanza variazionale K di Kolmogorov usando funzioni che non sono necessariamente funzioni di distribuzione probabilistica. Infatti O Maidin [27] propose proprio una tale misura del grado di somiglianza tra due melodie rappresentate come durate di altezza in funzione del tempo, come rappresentato in figura, dove l'asse x misura la durata temporale e l'asse y l'altezza delle note.

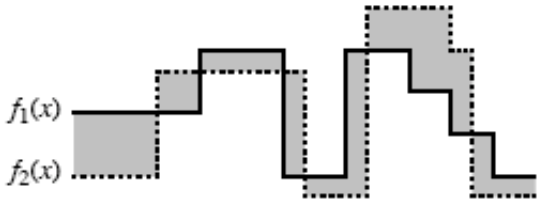


Fig. 2.19: due melodie rappresentate come funzioni rettilinee del pitch rispetto al tempo.

Nel contesto dell'estrazione di informazioni da melodie si potrebbe essere interessati nel calcolare l'area minima che stà tra le due curve traslandone una rispettivamente lungo l'asse x e/o y e mantenendo l'altra fissa. Aloupis et al. [41] hanno mostrato che il calcolo richiesto a partire da due melodie (catene poligonali) composte da n vertici è svolgibile in un tempo $O(n^2 \log n)$.

Utilizzeremo la misura K per confrontare ritmi utilizzando la rappresentazione cronotonica proposta da Gustafson [9]. Come esempio consideriamo il ritmo [x . . x x . x . . x x . x . . .] mostrato in figura in notazione cronotonica.

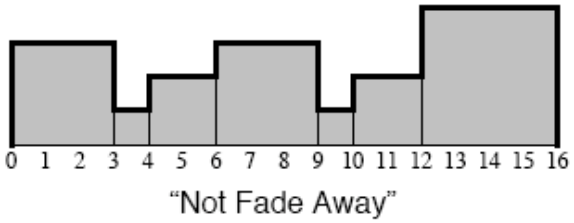


Fig. 2.20: il ritmo di Not Fade Away in notazione cronotonica.

Questa è la linea ritmica suonata dal tom-tom di Not Fade Away di Buddy Holly and the Crickets. Il disco venne pubblicato nel 1957 e divenne immediatamente un successo. E' una variante del famoso ritmo detto Bo Diddley Beat reso celebre da Bo Diddley nel suo successo "Bo Diddley" e dato da [x . x x x . x . . . x . x . . .]. Se sovrapponiamo le sequenze ritmiche di Not Fade Away e del pattern Shiko, come in figura, la misura K (la differenza delle aree) tra le due curve è 12. Un calcolo simile ma sovrapponendo Not Fade Away e il pattern Clave Son genera un valore di K pari a 10, mostrando che il ritmo di Not Fade Away è più simile al pattern Son che al pattern Shiko. Si vede quindi che nel contesto discreto è possibile calcolare facilmente K in tempo O(n).

2.2.5 Analisi geometrica di ritmi

Prendiamo in considerazione nuovamente la notazione musicale standard per il pattern clave Son illustrata in figura.

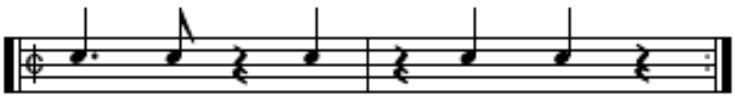


Fig. 2.21: notazione tradizionale del ritmo di clave Son

Chiediamoci ora se è possibile suonare il ritmo all'incontrario in modo che suoni esattamente uguale, partendo da uno a scelta dei suoi colpi. Osservando questo tipo di notazione non è semplice dare una risposta immediata a tale domanda.
La notazione box permette di rispondere più facilmente.

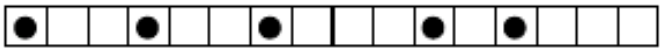


Fig. 2.22: notazione box del ritmo di clave Son

La risposta è sì se partiamo dal terzo colpo. Possiamo anche dire che il clave Son è un palindromo debole (in quanto necessità di sfasamento per risultare tale).

Una rappresentazione ancora migliore per tali ritmi che hanno carattere ciclico è data dalla rappresentazione a orologio, in cui ogni colpo è connesso al successivo da una linea andando a disegnare un poligono convesso. Questa rappresentazione non solo migliora notevolmente la visualizzazione del ritmo, ma si presta maggiormente all'analisi di carattere matematico.

Utilizzando questa rappresentazione su sei famosi ritmi di Clave ne risultano subito evidenti le principali caratteristiche.

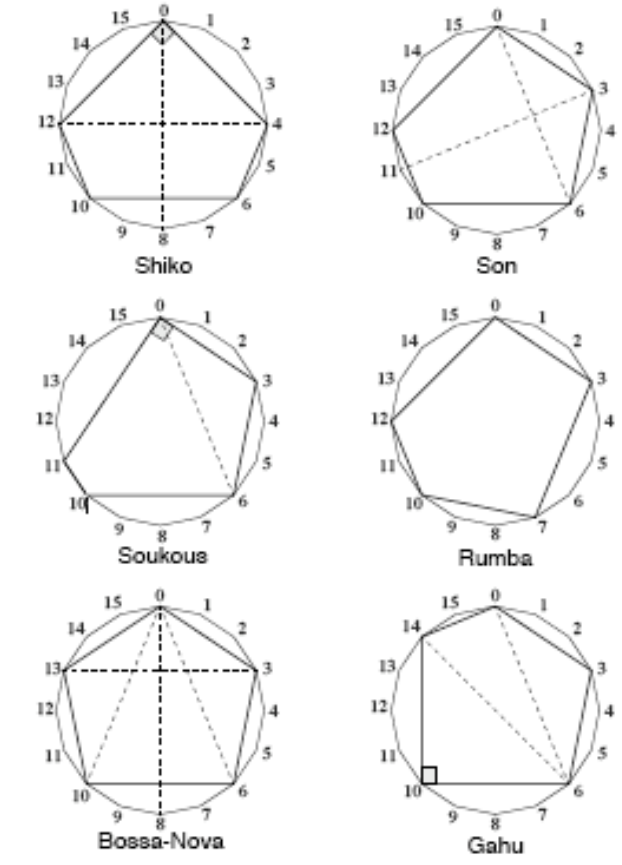


Fig. 2.23: i 6 ritmi di clave in 4/4 rappresentati come poligoni convessi inscritti in un cerchio.

Da notare che le linee tratteggiate della figura rappresentano o le basi di triangoli isosceli o un asse di simmetria speculare.

La rappresentazione a poligono convesso svela una grande varietà di proprietà geometriche discriminanti, utili per confrontare, analizzare e classificare pattern ritmici. Scoprire queste proprietà utilizzando la notazione classica o la notazione boxed non è altrettanto immediato. Ad esempio, dall'esame della figura risulta immediatamente ovvio che tre pattern (Shiko, Soukous e Gahu) tra i loro vertici ne contengono uno ad angolo retto, mentre gli altri tre (Son, Rumba e Bossa-Nova) no. E' da notare che i primi tre ritmi sono nati in Africa, mentre gli ultmi tre in America: Bossa-Nova in Brasile e Son e Rumba a Cuba. Dobbiamo interpretare un angolo retto come la caratteristica di un ritmo dal carattere più forte? Può essere la presenza di questo angolo retto collegata al diverso grado di popolarità dei ritmi africani rispetto a quelli americani?

Notiamo inoltre chs i pattern Shiko e Bossa-Nova sono palindromi. Suonano alla stessa maniera sia riprodotti lungo una direzione che nell'altra. Questo si può vedere dalla simmetria dei poligoni lungo la linea che và dalle posizione 0 alla 8. Inoltre vediamo che il pattern di Son è un palindromo debole, cioè esiste una posizione diversa da 0 da cui il ritmo suona uguale se suonato in un senso o nell'altro. In questo caso la posizione è al 3, dato che il poligono ha linea di simmetria tra le posizioni 3 e 11.

I pattern Shiko, Son e Soukous contengono un triangolo isoscele ciascuno. La presenza di un triangolo isoscele indica che il ritmo contiene due intervalli uguali consecutivi. Il pattern Gahu contiene due triangoli isosceli mentre il pattern Bossa-Nova ne contiene tre. Il pattern di Bossa-Nova è una struttura distribuita al grado massimo (maximally even set) [42]. Una struttura massimamente distribuita è tale quando gli elementi contenuti al suo interno sono distribuiti al grado massimo. Il pattern di Bossa-Nova contiene quattro intervalli lunghi tre e un intervallo di lunghezza quattro. All'opposto il pattern di Rumba non contiene alcun triangolo isoscele, nessun asse di simmetria e nessun angolo retto, ma le caratteristiche di questo ritmo sono comunque molto interessanti.

2.2.6 Misurare la complessità di un ritmo

Una caratteristica peculiare utilizzata per una varietà di applicazioni incluso il riconoscimento automatico dei ritmi è la complessità ritmica. Nel campo della Teoria dell'Informazione è stato dedicata grande attenzione alla misura della complessità delle sequenze [44] [43].

La complessità di "Lempel-Ziv"

Nel 1976 Lempel e Ziv [45] proposero una tipologia di misura della complessità di una sequenza finita all'interno del contesto della compressione dei dati. Seguendo un approccio innovativo la complessità di una sequenza finita viene determinata scorrendo la sequenza da sinistra verso destra alla ricerca delle più brevi sottosequenze (chiamate "words") che non sono ancora state rilevate durante la scansione. Ogni volta che una nuova word viene trovata entra viene raccolta insieme alle altre in un vocabolario. Quando la scansione è finita la dimensione del vocabolario determina la complessità della sequenza. Per sequenze cicliche come i ritmi da noi presi in cosiderazione risulta sufficiente analizzare una concatenazione di due sole istanze del pattern ritmico in quanto ripetendo ulteriormente la concatenazione non vengono trovate nuove parole da aggiungere al vocabolario. Come esempio consideriamo il pattern di clave Son. In figura 2.24 è mostrato il ritmo espresso in notazione binaria.

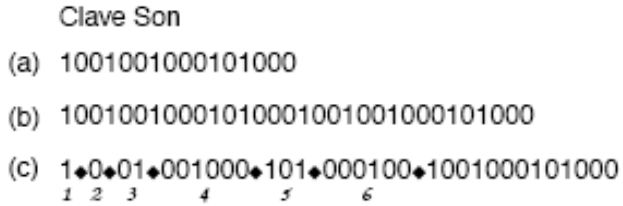


Fig. 2.24: il calcolo della complessità di Lempel-Ziv sul ritmo di clave Son.

Ripetendo il pattern una seconda volta otteniamo un pattern composto di 32 bit.
In figura 2.24 è quindi mostrata l'individuazione di ogni nuova parola, assegnando ad ognuna un proprio indice numerico. Nel pattern clave Son, seguendo questo processo, vengono generate 6 nuove sottosequenze e quindi il grado di complessità di Lempel-Ziv è pari a 6.

Rhythm Complexity Measures

	Lempel-Ziv	Pressing	Metric
Shiko	5	6	2
Son	6	14.5	4
Soukous	6	15	6
Rumba	6	17	5
Bossa-Nova	5	22	6
Gahu	5	19.5	5

Fig. 2.25: il confronto tra tre misure di complessità di un ritmo.

Confrontando i risultati ottenuti per i sei diversi ritmi di Clave in figura 2.25 notiamo che questo tipo di misura non è particolarmente utile. Non esiste praticamente variabilità nei risultati: tutti sono o 5 o 6. Inoltre questi risultati non comunicano nulla a chi è esperto nell'insegnamento o nell'esecuzione di questi ritmi. Ad esempio il pattern Shiko è il più semplice dei sei ritmi e il pattern Gahu è uno dei più complessi, sia da riconoscere che da suonare, ma la complessità di Lempel-Ziv ha lo stesso valore per entrambi. Questa misura legata al campo della teoria dell'informazione non rispecchia bene il modo in cui l'essere umano sente la complessità percettiva, cognitiva eds esecutiva dei ritmi.

La complessità cognitiva di un ritmo

Jeff Pressing ha proposto una misura della complessità cognitiva di un ritmo basata su principi psicologici e il grado di sincope presente nel ritmo a differenti livelli di pulsazione [46]. In figura mostriamo la complessità cognitiva calcolata da Pressing per i dieci pattern di quattro unità creati con una nota sola e con due. Se prendiamo i pattern di 16 bit dei ritmi di Clave, li dividiamo in 4 unità di 4 bit ciascuna, e sommiamo le complessità di Pressing delle quattro corrispondenti unità otteniamo il valore di complessità di Pressing dei sei ritmi di Clave. Per esempio, il pattern di Shiko consiste nella concatenazione dei pattern [1 0 0 0], [1 0 1 0], [0 0 1 0] e [1 0 0 0]. Facendo riferimento alla figura otteniamo le complessità 0, 1, 5 e 0 per un totale di 6.

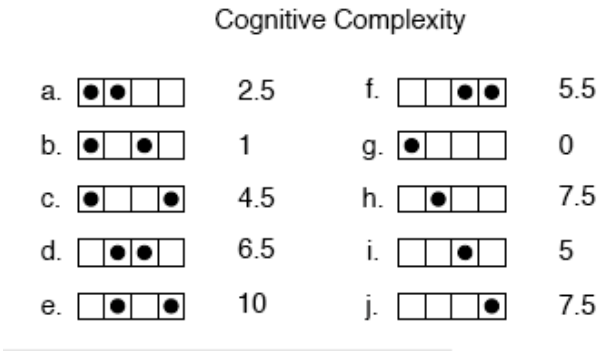


Fig. 2.26: la complessità cognitiva di 10 pattern base.

D'altro canto il pattern di Rumba genera una complessità cognitiva di Pressing di 4.5 + 7.5 + 5 + 0 = 17. L'esame della complessità cognitiva di Pressing dei 6 ritmi di Clave fornisce un'informazione più precisa rispetto alla misura di complessità di Lempel-Ziv. Ogni risultato è differente e la varianza è abbastanza ampia con un intervallo che và dal valore 6 del pattern Shiko al valore 22 del pattern Bossa-Nova. I risultati rispecchiano inoltre le esperienze legate all'insegnamento e all'esecuzione dei pattern ritmici. Il pattern di Shiko è semplice, quello di Rumba è più complesso di quello di Son, e quelli di Bossa-Nova e Gahu sono i ritmi più difficili tra tutti e sei.

La metricità e la complessità metrica

Lerdahl e Jackendoff [47] hanno proposto un costrutto chiamato "struttura metrica" per descrivere l'organizzazione psicologica temporale dei pattern ritmici ad ogni livello metrico. La loro struttura metrica per l'intervallo di 16 unità relativo ai pattern di Clave è mostrata in figura.

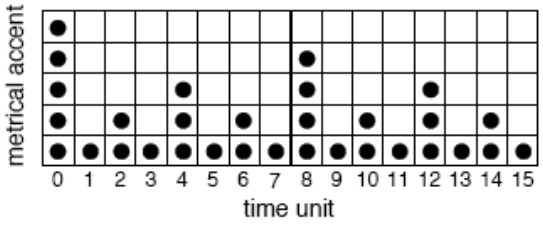


Fig. 2.27: la struttura metrica su una battuta divisa in 16 parti.

La struttura definisce una funzione che mappa l'indice dell'unità temporale con l'intensità relativa dell'accento metrico. Il modo più semplice per descrivere la funzione è sommare le intensità delle diverse unità temporali. Al primo livello di accento metrico un valore di intensità è aggiunto per ogni unità temporale a partire da quella con valore 0. Al secondo livello di accento metrico un valore di intensità è aggiunto ogni due unità temporali a partire da 0. Al terzo livello di accento metrico un valore di intensità è aggiunto ogni quattro unità temporali a partire da 0. Si prosegue così raddoppiando la lunghezza dell'intervallo per ogni livello di intensità successivo. In questo modo l'unità temporale 8 ha un valore di intensità pari a quattro e l'unità temporale 0 raggiunge un valore di intensità pari a cinque. Di conseguenza le unità temporali 2, 6, 10 e 14 sono considerate ad intensità debole, le unità temporali 4 e 12 sono indicate come di intensità media, l'unità temporale 8 è di intensità maggiore e l'unità temporale 0 è la più forte di tutto l'insieme.

Una nuova misura di complessità di un ritmo può essere definita basandosi sul concetto appena visto di metro. Definiamo innanzitutto una misura della forza metrica complessiva di un ritmo e chiamiamola metricità. Essa consiste semplicemente della somma dei valori di intensità di tutte le unità temporali del ritmo. Ad esempio, il pattern di Clave Son ha i colpi sulle unità temporali 0, 3, 6, 10 e 12. Gli accenti metrici corrispondenti a queste unità temporali sono 5, 1, 2, 2 e 3, rispettivamente. Quindi la metricità del pattern di Clave Son, secondo la struttura gerarchica di Lerdahl e Jackendoff è pari a 13. Dato che questa struttura metrica possiede una semplice gerarchia altamente strutturata, la funzione di metricità defnita qui è più una misura di semplicità del pattern, piuttosto che di complessità. E' quindi inversamente proporzionale a ciò che sarebbe chiamata "complessità metrica". Il valore massimo della metricità considerando cinque colpi non può superare il valore 17 e quindi la complessità metrica sarà definita come 17 meno la metricità. Quindi la complessità metrica del pattern di Clave Son, ad esempio, è di 17-13 = 4.

Nonostante la struttura definita da Lerdahl e Jackendoff sia basata sulla musica occidentale/europea rimane comunque un'interessante punto di vista da cui osservare i ritmi di origine africana e afro-americana. Comunque, anche se questa misura ha una valenza psicologica bassa e un carattere non universale, al peggio è altrettanto valida come misura matematica oggettiva quanto la complessità di Lempel-Ziv e può essere applicata come metodo di estrazione di caratteristiche ritmiche per la classificazione automatica di ritmi. L'esame della complessità metrica dei sei pattern di Clave rivela che la misura è decisamente migliore di quella della complessità di Lempel-Ziv. La varianza è accettabile, il pattern di Shiko possiede il valore

minimo pari a 2, quello di Rumba (5) ha un valore maggiore di quello di Son (4) e quello di Bossa-Nova ha il valore più alto, 6. Comunque, la misura della complessità metrica non è altrettanto valida quanto quella della complessità cognitiva di Pressing. I pattern di Bossa-Nova e Gahu sono più difficili da suonare di quello di Soukous, per esempio, mentre la complessità metrica del pattern di Soukous lo indica come maggiormente complesso del pattern di Gahu.

2.2.7 Analisi combinatoria di ritmi

L'applicazione del calcolo combinatorio all'analisi della musica non è una novità. Tuttavia, quasi sempre questo tipo di analisi è stato applicato alla caratteristica verticale della musica (la scala tonale) piuttosto che a quella orizzontale (la scala temporale) [48] [42]. Fanno eccezione a questa tendenza i due paper di Haak [49] [50] e quello di London [51].
Steve Reich [52] ha composto un pezzo di notevole interesse chiamato Clapping Music per due persone che battone le mani. Entrambi gli esecutori battono esattamente lo stesso rtimo di dodici unità mostrato in figura.

Fig. 2.28: la sequenza usata da Steve Reich in Clapping Music

Un esecutore esegue il pattern in maniera continuativa dall'inizio alla fine del pezzo, mentre l'altro esecutore, dopo aver eseguito la sequenza 12 volte, ritarda la sequenza di una unità temporale. Il secondo esecutore continua a ritardare la seuenza di una unità temporale nella stessa direzione ogni volta che il pattern viene esguito 12 volte. Il pezzo finisce con il secondo esecutore che ritorna in fase con il primo. Quindi gli ultimi 12 pattern combinati risultano esattamente uguali ai primi dodici essendo entrambi suonati perfettamente in fase.
Steve Reich ha composto Clapping Music con 8 colpi (battiti di mani). Il numero di modi in cui possiamo scegliere 8 unità temporali in una battuta composta da 12 è (12!)/(8!)(4!) = 495. Joel Haak ha proposto l'interessante domanda su come Reich abbia scelto il pattern all'interno di questi 495 possibili. Li ha ascoltati tutti scegliendopoi quello che preferiva? Haak suggerisce una risposta matematica alla domanda. Se prendiamo in considerazione che un brano deve iniziare con un colpo e non con una pausa, se consideriamo equivalenti le permutazioni cicliche di un pattern, se poniamo che durante l'esecuzione del brano il pattern risultante dalla combinazione dell'esecuzione dei due performer non si deve ripetere, e infine se non ammettiamo la ripetizione consecutiva del numero dei colpi contenuti tra due pause consecutive, allora solo 2 dei 495

pattern possibili soddisfano queste richieste. Il primo è il pattern [3,2,1,2] scelto da Reich. Il secondo è il pattern [4,1,2,1] in figura 2.29.

Fig. 2.29: la seconda sequenza trovata da Joel Haak

Haak non ci fornisce un criterio per scegliere tra uno di questi due ritmi. Ma se vogliamo minimizzare il numero massimo di colpi consecutivi, allora otteniamo il pattern di Reich. In alternativa possiamo richiamare il criterio di sistema massimamente distribuito [42] per arrivare al pattern scelto da Reich.

Permutazioni e sistemi multipli

I pattern di Clave che abbiamo visto precedentemente sono composti con 5 colpi ciascuno in una struttura ritmica di 16 unità. Il numero di modi in cui possiamo scegliere 5 colpi in una struttura composta da 16 unità temporali è (16!)/(5!)(11!) = 4368, un numero molto grande di pattern. La maggior parte di questi pattern molto probabilmente non sono utili per generare ritmi potenti e coinvolgenti. Come possiamo ridurre questo grande numero ad un sottoinsieme ridotto di pattern interessanti? Consideriamo innanzitutto che in tutti i pattern di Clave, tranne il pattern di Soukous, l'intervallo minimo tra un colpo e il successivo e quello massimo sono rispettivamente 2 e 4. Aggiungiamo quindi questa restrizione all'insieme delle possibili soluzioni. Prendiamo in considerazioni il pattern di Clave Son rappresentato come sequenza binaria [1 0 0 1 0 0 1 0 0 0 1 0 1 0 0 0] e il suo vettore intervallare (2 2 3 1 3) corrispondente al numero di unità temporali saltate (zeri) tra un colpo e il successivo (uni). Quante permutazioni esistono del pattern (2 2 3 1 3)? Da notare che ora questi sono sistemi multipli in quanto la ripetizione degli elementi è ammessa. Abbiamo quindi cinque oggetti appartenenti a tre classi differenti: 1 di classe uno, 2 di classe due e 2 di classe tre. Quindi il numero totale di permutazioni differenti di (2 2 3 1 3) è (5!)/(1!)(2!)(2!) = 30.

Se proviamo a suonare queste permutazioni ci accorgiamo che tutte danno ottimi risultati. Tra questi 30 troviamo il pattern di Rumba, quello di Gahu, come la versione contraria del pattern di Son, di Rumba e Gahu. Se un ritmo si può ottenere da un altro con una permutazione del suo

vettore intervallare si dirà che i due ritmi appartengono alla stessa classe combinatoria intervallare. Quindi i pattern di Son (2 2 3 1 3), Rumba (2 3 2 1 3) e Gahu (2 2 3 3 1) appartengono alla stessa classe combinatoria intervallare, mentre i pattern di Shiko (3 1 3 1 3), Soukous (2 2 3 0 4) e Bossa-Nova (2 2 3 2 2) appartengono ciascuno ad una propria classe combinatoria intervallare.

Ritornando ai 30 ritmi della classe combinatoria intervallare Son-Rumba-Gahu, ed escludendo il pattern di Son suonato al contrario in quanto è un palindromo debole, i rimanenti 26 ritmi hanno una strana somiglianza con i pattern di Son, Rumba e Gahu, ma suonano più moderni. Ognuno di questi non stonerebbe all'interno di pezzi composti ai giorni nostri. Questa tecnica combinatoria intervallare suggerisce un fruttuoso approccio alla generazione automatica di nuovi ritmi.

3

AUTOCOMPOSIZIONE DI PATTERN RITMICI

3.1 La struttura MVC

Il codice è stato sviluppato secondo i paradigmi della struttura MVC (Model-View-Controller). La struttura MVC divide l'applicazione in tre categorie principali di oggetti e ottimizza l'interazione tra di essi. Ogni categoria assegna ai suoi oggetti ruoli specifici e definisce il modo in cui essi interagiscono.

Model

Gli oggetti nella categoria Model costituiscono il centro computazionale dell'applicazione. Tutti gli oggetti che si occupano del processing dei valori in ingresso forniti dall'utente sono inseriti nel Model. Gli oggetti nel Model si occupano esclusivamente di elaborare dati in input e fornire i risultati in output. In questa applicazione il modello è costituito da tutti gli oggetti che si occupano di calcolare il posizionamento dei colpi all'interno della griglia temporale.

View

Questi oggetti si occupano di far comunicare il programma con l'utente. In questa categoria rientrano tutti gli oggetti grafici che servono all'immissione dei dati da parte dell'utente (come i bottoni, gli slider, ecc...) e tutti quelli che il programma usa per mostrare i risultati all'utente

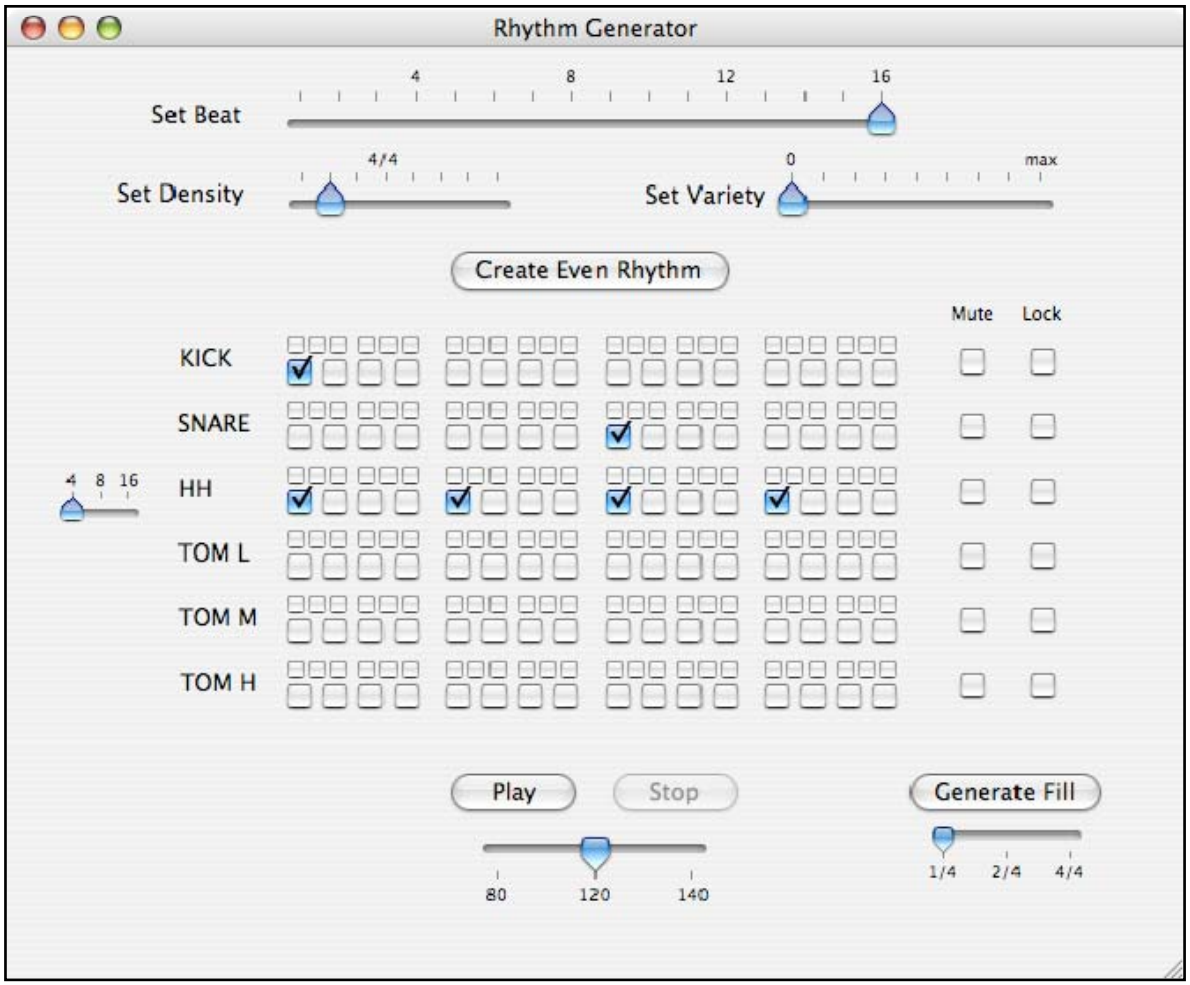
(campi di testo, disegni, ecc...). In questa applicazione gli oggetti della view sono gli slider, i bottoni e le checkbox presenti sull'interfaccia grafica creata tramite Interface Builder.

Controller

Gli oggetti Controller sono il trait d'union tra il centro elaborazione dati e l'interfaccia utente. Un Controller prende i dati in ingresso provenienti da un oggetto della View e li smista agli appropriati oggetti del Model. Provvede inoltre a prendere i risultati calcolati dal Model e a indirizzarli verso gli oggetti della View corretti. Alcuni risultati saranno mostrati in un campo di testo, altri creeranno dei disegni sullo schermo, altri modificheranno lo stato di bottoni o slider.
I Controller sono il centro di smistamento dei dati immessi dall'utente e di quelli ritornati dall'elaboratore.
Vediamo ora più nello specifico come sono state implementate queste tre categorie all'interno del programma.

3.2 L'interfaccia utente (view)

L'interfaccia grafica presenta una struttura abbastanza semplice e si suddivide in tre parti principali: i parametri per la creazione del ritmo, la griglia di visualizzazione del pattern creato, il player.



3.2.1 I parametri per la creazione del ritmo

I parametri che definiscono la creazione del ritmo sono modificabili tramite quattro slider.

Set Beat

Lo slider Set Beat permette di variare la dimensione della battuta, aumentando o diminuendo il numero di sedicesimi inclusi. In questo modo il programma rende possibile la creazione di una notevole varietà di ritmi.

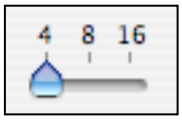
Set Density

Grazie a questo slider si ha la possibilità di definire il numero di colpi di cui è costituito il ritmo di base. Alta densità corrisponde ad un elevato numero di colpi, bassa densità ad un numero ridotto. Il minimo valore impostabile sullo slider è uno, ed il massimo è otto. Il numero di colpi impostato viene distribuito equamente tra gli strumenti cassa e rullante; quindi con il valore 1 avremo un unico colpo di cassa, con il valore 2 avremo un colpo di cassa e uno di rullante, con il valore 3 avremo il primo colpo di cassa, il secondo di rullante e il terzo di cassa, e via così fino a 8 colpi dove avremo 4 colpi di cassa e 4 di rullante tra di loro alternati. In questo modo, impostando il valore a 4 (e mantenendo lo slider Set Beat al massimo della sua escursione), si otterrà un tipico ritmo in quattro quarti. Nel caso il ritmo sia composto da un numero di colpi dispari l'ultimo colpo varierà casualmente qualità timbrica. Ad esempio l'ultimo colpo di un ritmo composto di tre colpi sarà di cassa o di rullante in maniera casuale, dando alla creazione dei ritmi un ulteriore grado di variabilità.

Set Variety

Il parametro Set Variety permette di regolare il grado di "equilibrio" presente all'interno del gruppo degli intervalli temporali che compongono il ritmo. Si considera come grado zero della varietà la massima distribuzione dei colpi all'interno della battuta. La varietà in questo caso è ridotta al minimo in quanto tutti i colpi sono equispaziati temporalmente e quindi la percezione di variazioni da parte dell'ascoltatore è minima. Per fare un esempio, se si è scelto di inserire 4 colpi tramite il parametro Set Density, la varietà minima sarà quella in cui i colpi sono distribuiti sui beat 1-5-9-13.
Aumentando il valore della varietà in pratica si chiede al programma di sfasare temporalmente i colpi inseriti in grado sempre maggiore rispetto alla loro posizione di massima distribuzione.
Questo parametro non è selezionabile all'avvio del programma in quanto si vuole che l'utente generi inizialmente una distribuzione ritmica a grado zero, e solo dopo possa variarne la struttura. Il parametro dipende dai primi due (Set Beat e Set Density) in quanto il numero massimo di spostamenti che i colpi possono fare all'interno della battuta è funzione della dimensione della battuta e del numero di colpi presenti al suo interno.

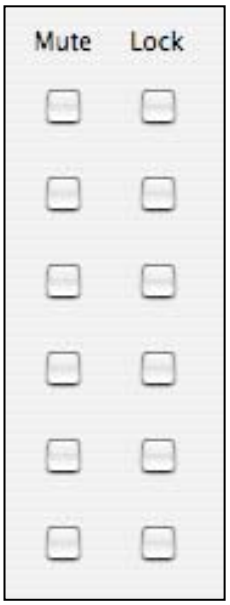
4-8-16



Questo parametro permette di definire la quantità dei colpi all'interno del pattern di charleston. Il valore 4 fà inserire al computer quattro colpi di charleston, il valore 8, otto e sedici per l'ultimo valore.

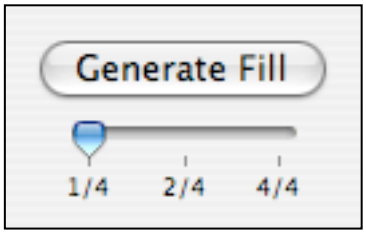
Altre Funzioni

Tasti Lock e Mute



I tasti Lock permettono all'utente di evitare che il pattern di un certo strumento venga modificato. Premendo il tasto Lock associato alla cassa, ad esempio, si eviterà che il pattern della cassa venga modificato quando si preme il tasto Create Rhythm.
Attraverso i tasti Mute l'utente può impedire che i pattern di alcuni strumenti vengano riprodotti, in modo da poter ascoltare le diverse componenti timbriche di un pattern composto.

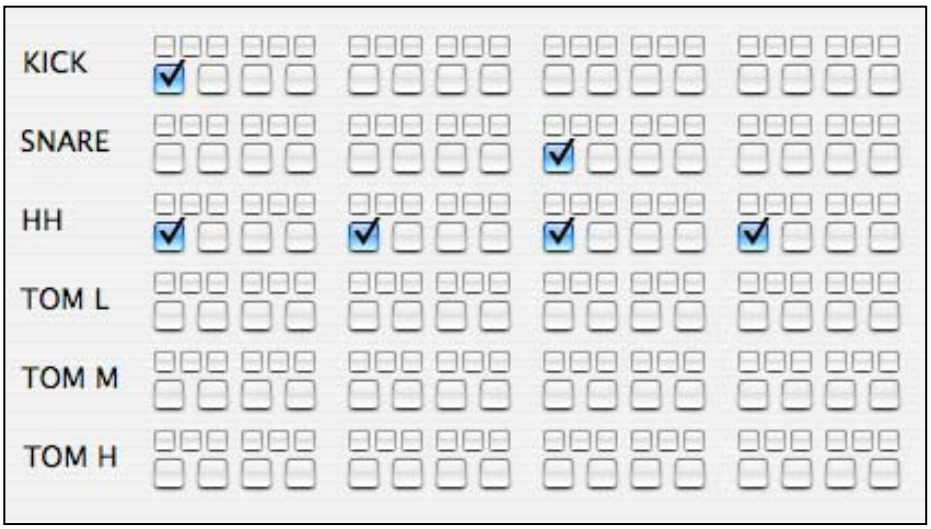
Il generatore di fill



Nella parte in basso a destra dell'interfaccia ho inserito la funzione di generazione automatica di fill. Con fill si intendono quelle variazioni ritmiche solitamente costituite da fitte sequenze di

colpi sui vari strumenti della batteria, che vengono utilizzate per spezzare la monotonia del ritmo portante e/o evidenziare in maniera ritmica la struttura compositiva di un brano musicale.
Il tasto Generate Fill crea una sequenza di colpi distribuita su tutti i pezzi della batteria: cassa, rullante, charleston, timpano, tom medio e tom piccolo. Il modo in cui i colpi sono distribuiti è basato su un algoritmo di distribuzione casuale, tarato su valori statistici a loro volta ricavati da strutture di fill classiche. Attraverso lo slider con i tre valori: un quarto, due quarti e quattro quarti è possibile decidere la durata del fill. Maggiore sarà il valore di durata, maggiore sarà la variazione introdotta alla fine del ritmo portante generato tramite il tasto Create Rhythm.

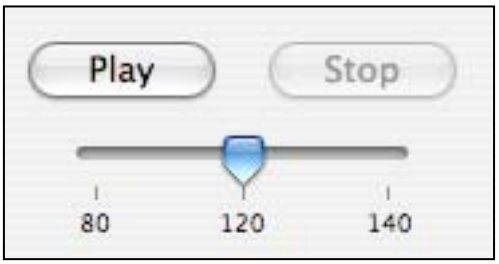
3.2.2 La griglia di visualizzazione del pattern



La griglia che permette di visualizzare il pattern creato è costituita da 12 serie di checkbox. Per ogni strumento che compone la batteria acustica: Kick (cassa), Snare (rullante), Hi-Hat (charleston), Floor o Low Tom (timpano), Mid Tom (tom medio), High Tom (tom piccolo), ho creato una doppia serie di checkbox. Le checkbox di dimensioni maggiori rappresentano i sedicesimi della battuta, mentre quelle di dimensioni inferiori servono all'inserimento di valori terzinati all'interno del ritmo. Questo torna utile soprattutto nella creazione dei fill ritmici quando le terzine giocano un ruolo importante nel dare l'impressione di una variazione/accelerazione momentanea del ritmo; è comunque possibile inserire valori terzinati manualmente all'interno di un ritmo creato solo sui sedicesimi per sperimentare soluzioni inusuali o creare particolari micro variazioni sul ritmo base.

Ogni qualvolta un colpo di uno strumento viene collocato in una posizione la checkbox relativa modifica il suo stato da unchecked (non spuntata) a checked (spuntata). Quando il numero di sedicesimi che costituiscono il pattern viene modificato tramite il parametro Set Beat le checkbox in più mutano il loro stato in 'trasparente' in modo che i relativi sedicesimi non siano più visualizzabili sulla griglia. Quando si avvia la riproduzione dell'audio per ascoltare il pattern le checkbox mutano una di seguito all'altra il loro stato da 'acceso' a 'spento' in modo che l'utente possa visualizzare lo scorrere del tempo all'interno del pattern.

3.2.3 Il Player



Nella parte inferiore dell'interfaccia si trova il Player. I bottoni Play e Stop permettono rispettivamente di avviare e fermare la riproduzione sonora del pattern.
Grazie allo slider sottostante è possibile regolare la velocità di riproduzione del pattern. I tre valori possibili sono 80, 120 e 140 Bpm (beats per minute - battiti al minuto).

3.3 Gestione input-output e algoritmi di definizione dei pattern (controller e model)

La classe `GUIController` si occupa dell'acquisizione dei parametri impostati dall'utente tramite interfaccia grafica e della rappresentazione su questa dei ritmi elaborati dagli algoritmi di autocomposizione.

3.3.1 L'impostazione iniziale dell'interfaccia

Il primo metodo che compare all'interno della classe `GUIController` è il metodo `awakeFromNib.`

Il metodo `awakeFromNib` viene chiamato quando tutti gli oggetti contenuti nel file Nib sono stati caricati e tutte le connessioni tra il codice e gli oggetti grafici sono state create. Inserire messaggi diretti agli oggetti dell'interfaccia in questo metodo permette di personalizzare il modo in cui appare l'interfaccia utente all'apertura.

Nel nostro metodo `awakeFromNib` vengono disabilitate le parti della GUI non necessarie al momento dell'apertura dell'applicazione, cioè quelle che l'utente non utilizzerà prima di aver generato almeno un ritmo. Le parti dell'interfaccia disabilitate all'avvio sono: lo slider `variety`, il bottone `createRtmVariation` (nascosto), i bottoni di Lock, la sezione Player e la sezione di generazione dei fill.

All' interno di questo metodo ho anche raggruppato tutte le allocazioni necessarie alla creazione degli array di checkbox che rappresentano graficamente il loop ritmico. Inserire queste allocazioni nel metodo `awakeFromNib` piuttosto che nel metodo `createRhythm` consente di allocare la memoria per gli array di bottoni una volta sola e non ogni volta che si crea un ritmo nuovo, evitando riallocazioni inutili. Il metodo `awakeFromNib` alloca quindi la memoria per gli array di bottoni che fungeranno da tracce per i vari strumenti della batteria: Kick, Snare, Hi-Hat, low Tom, mid Tom e high Tom.

Alla fine del metodo `awakeFromNib` viene infine inizializzato un valore booleano che serve a descrivere al programma stesso lo stato in cui si trova. All'apertura lo stato sarà: "nessun ritmo ancora creato" (`rhythmCreated = NO`).

3.3.2 La creazione del ritmo

Le prima operazione che l'utente compie non appena aperta l'applicazione sarà impostare i valori per la definizione di un ritmo e crearlo. Ogni possibile azione richiama un apposito metodo appartenente alla classe `GUIController`.

Impostazione della durata del loop (Set Beat)

La durata del loop è impostata dallo Slider Set Beat. Questo slider può assumere i valori da 1 a 16 e determina il numero di sedicesimi che compongono il loop.
La variazione del valore dello Slider `beatNum` richiama il metodo `beatChange.` Il metodo `beatChange` si occupa di reimpostare i valori degli slider utilizzati per determinare il numero di colpi di cassa, rullante e charleston.
Il primo slider controllato da `beatChange` è `hitNum`. Nel caso la lunghezza del loop venga impostata ad un valore inferiore a 8 il numero di valori disponibili sullo slider `hitNum` viene modificato in modo da non abilitare la scelta di un numero di colpi superiore al numero di sedicesimi che compongono il loop. Ad esempio un valore di lunghezza di loop di 6 sedicesimi modificherà lo slider `hitnum` in modo da rendere possibile la generazione di ritmi composti da non più di sei colpi.
La seconda operazione compiuta da `beatChange` è sullo slider che controlla il numero di colpi del charleston. Lo slider del charleston è in realta composto di tre slider di lunghezza decrescente che vengono resi visibili uno per volta a seconda del valore di `beatNum`. Quando il valore di `beatNum` è pari a 16 viene visualizzato lo slider contenente i tre valori 4, 8 ,16. Quando il valore di `beatNum` è compreso tra 16 e 8 (incluso) viene visualizzato lo slider contenente i due valori 4 e 8. Infine per valori di `beatNum` inferiori a 8 lo slider visualizzato è composto di un unico valore, pari a 4. In questo modo per l'utente non è possibile inserire un numero di colpi di charleston maggiore della lunghezza del loop.
Dopo aver visualizzato lo slider del charleston corretto il metodo `beatChange` passa a impostare la visualizzazione della griglia di checkbox. Diminuendo il valore di `beatNum` diminuiscono di conseguenza le checkbox visualizzate. Le checkbox che superano il valore impostato con `beatNum` vengono rese invisibili utilizzando il metodo `setTransparent`.

E' qui che entra in gioco la variabile booleana `rhythmCreated`. Nel caso in cui il valore dello slider venga variato prima che almeno un ritmo sia stato generato il metodo `beatChange` si limita a fare le suddette operazioni. Quando invece almeno un ritmo è stato generato, la variazione di valore dello slider `beatNum`, oltre a controllare il valore degli altri slider, genererà un ritmo (richiamando il metodo `createEvenRhythm`).

Impostazione della quantità di colpi (Set Density e 4/8/16)

Lo slider `hitNum` (Set Density) permette all'utente di determinare il numero di colpi di cui è costituito il pattern principale, ovvero quello costituito da cassa e rullante. L'utente può decidere di inserire un numero di colpi compreso tra 1 e 8, che verrà uniformemente distribuito tra cassa e rullante (vedremo meglio fra poco).
Ogni qualvolta si imposta lo slider `hitNum` lo slider `variety` (Set Variety) viene disabilitato in modo che sia possibile generare solo pattern a massima distribuzione. L'utente dovrà sempre prima generare un pattern massimamente distribuito, e solo dopo potrà variare la distribuzione di questo pattern con lo slider `variety`. Inoltre l'impostazione di questo slider gestisce la visualizzazione del bottone principale dell'applicazione: toccando o variando il valore dello slider `hitNum` il bottone di creazione dei ritmi passa dallo funzione di creazione di variazioni del ritmo base a quella di creazione di un ritmo base (la scritta sul tasto muta da "Create Rhythm Variation" a "Create Even Rhythm").
Attraverso lo slider a fianco della traccia del charleston è possibile impostare quanti colpi di questo strumento suonare. E' possibile inserire 3 valori: 4, 8, 16, la cui disponibilità dipende dall'impostazione dello slider `beatNum` (come descritto sopra). Ogni volta che si reimposta questo slider il pattern di charleston viene rigenerato.

Generazione del ritmo

Dopo aver impostato i parametri per la generazione del ritmo all'utente non rimarrà altro da fare che generare il ritmo. Questo avviene premendo il tasto `createEvenRtm` (Create Even Rhythm).
Il tasto `createEvenRtm` richiama il metodo `createEvenRhythm`, il quale si occupa di:
1) reimpostare la GUI abilitando le nuove funzionalità,

2) cambiare lo stato dell'applicazione da "nessun ritmo ancora generato" a "almeno un ritmo è stato già generato",
3) generare un pattern massimamente distribuito a partire delle impostazioni dell'utente,
4) generare un valore casuale utile all'assegnazione dell'ultimo colpo dei pattern dispari allo strumento cassa o rullante (vedremo meglio in seguito),
5) lanciare la generazione dei pattern dei singoli strumenti in base tenendo conto dello stato dei bottoni di Lock (disabilitati all'avvio),
6) calcolare il massimo valore di sfasamento ammesso per il pattern cassa+rullante appena creato e impostare lo slider `variety` di conseguenza.

1 - Reimpostazione della GUI

In seguito alla pressione del tasto `createEvenRtm` la GUI viene reimpostata abilitando lo slider `variety`, i tasti di Lock e il generatore di fill. Infine viene verificato lo stato del Player andando ad abilitare il tasto play e lo slider `bpm` nel caso in cui il tasto di stop sia disabilitato.

2 - Passaggio di stato dell'applicazione a: "almeno un ritmo è stato creato"

Questa variazione della variabile booleana `rhythmCreated` serve a gestire la funzionalità dello slider `beatNum`. Quando questa variabile è YES lo slider `beatNum` attiva la funzione di ricreazione automatica del pattern.

3 - Generazione del pattern massimamente distribuito

Viene generato un pattern massimamente distribuito in funzione delle impostazioni dell'utente. Il metodo di generazione del pattern massimamente distribuito si chiama `createEvenPattern` ed appartiene alla classe `patternCreator`, che contiene i metodi fondamentali per la generazione dei pattern. Tra poco mi occuperò della descrizione di questo metodo.

4 - Variabile casuale utile per l'assegnazione dell'ultimo colpo dei pattern dispari

I pattern massimamente distribuiti possono avere un numero di colpi tra 1 e 8. Quando questo numero è pari i colpi vengono equamente distribuiti tra cassa e rullante in maniera sequenziale. Ad esempio se il numero di colpi è pari a 4, il primo colpo verrà assegnato al pattern di cassa, il secondo al rullante, il terzo alla cassa e il quarto di nuovo al rullante. Seguendo questa logica

quando il numero di colpi è dispari l'ultimo colpo sarebbe sempre suonato dalla cassa. La variabile casuale generata qui permette di avere colpi finali di patter dispari anche sul rullante. In questo modo, mentre i pattern pari avranno sempre la stessa distribuzione di colpi tra cassa e rullante, i pattern dispari no, in quanto l'ultimo colpo varierà alternerà la sua assegnazione in maniera casuale tra cassa e rullante. Questo si può vedere facilmente scegliendo un valore di `hitNum` dispari e premendo diverse volte il tasto `createEvenRtm`.

5 - generazione dei pattern singoli di ogni strumento
In questa parte del metodo `createEvenRhythm` vengono chiamati i metodi per la generazione dei pattern di ogni singolo componente del ritmo. I pattern dei singoli strumenti vengono generati solo nel caso in cui i corrispettivi tasti di Lock non siano premuti. Descriverò meglio la generazione dei singoli pattern di seguito.

6 - calcolo del massimo sfasamento
l'ultima operazione che compie il metodo `createEvenRhythm` è quella di calcolare il massimo sfasamento ammissibile per il pattern cassa+rullante appena creato. Il massimo sfasamento viene ottenuto prendendo il valore minimo tra i due valori `maxKickShift` (massimo sfasamento ammesso per il pattern di cassa) e `maxSnareShift` (massimo sfasamento del pattern di rullante). Questi due valori vengono calcolati al momento della creazione dei singoli pattern di cassa e rullante.
In base poi a questo valore massimo di sfasamento viene impostato lo slider `variety`, il quale permetterà all'utente di variare le posizioni dei colpi nel pattern massimamente distribuito.

La creazione del pattern massimamene distribuito

Il metodo che si occupa della creazione dei pattern massimamente distribuiti è il metodo `createEvenPattern` , contenuto nella classe `PatternCreator`. Il metodo crea un pattern massimamente distribuito a partire da due valori: la lunghezza del loop (`aBeatNumValue`) e il numero di colpi che deve contenere (`aHitNumValue`).
La prima operazione compiuta dal metodo è la creazione di un array di sedici colpi. Vengono allocati 16 oggetti di tipo `Hit` (il numero massimo di colpi che può contenere un pattern

massimamente distribuito in questa applicazione) e vengono inseriti in un array.
Ogni oggetto `Hit` contiene al suo interno le variabili identificative che vediamo qui di seguito (e i relativi metodi per acquisirne e mutarne il valore):
- `order`: l'ordinalità del colpo (primo, secondo, terzo.... colpo)
- `position`: la posizione all'interno dell'array di sedici elementi
- `center`: la posizione del colpo quando è massimamente distribuito
- `eccentricity`: il numero di spazi tra la posizione corrente del colpo e la sua posizione massimamente ditribuita
- `intervalDuration`: la durata dell'intervallo di silenzio successivo al colpo.

In seguito alla creazione dell'array di colpi vengono definiti i bordi di tale array ponendo un oggetto `evenHit0` come limite minimo (`position = 0`) e assegnando la posizione di limite massimo (`position = beatNumValue+1`) al colpo successivo all'ultimo appartenente all'array.
Ora possiamo procedere con la creazione degli intervalli di silenzio tra un colpo e l'altro.
Sapendo il numero di spazi disponibili all'interno del loop, ad esempio 16, e il numero di colpi che dobbiamo inserire, mettiamo 4, risulta semplice calcolare il valore dell'intervallo di silenzio tra un colpo e l'altro: numero degli spazi/numero dei colpi = durata dell'inervallo (16/4 = 4). Questo quando il numero dei colpi è sottomultiplo di quello degli spazi. Quando non è così scopriamo che redistribuendo equamente tra i colpi il resto ottenuto dalla divisione di cui sopra otteniamo ugualmente pattern massimamente distribuiti.
Facciamo un esempio: prendiamo sempre 16 spazi disponibili, ma stavolta poniamo di voler inserire 6 colpi. Facendo la divisione otteniamo che il valore di intervallo è pari a 2, con resto 4. Abbiamo quindi 6 intervalli di valore 2 e un resto di 4 spazi. Non dobbiamo far altro che redistribuire il resto tra gli intervalli che abbiamo già. Dando uno spazio a ciascun intervallo fino ad esaurire gli spazi rimanenti (il resto 4) avremo, anzichè 6 intervalli di valore 2, 4 intervalli di valore 3 e 2 intervalli di valore 2, per un totale di sedici spazi occupati.
L'algoritmo di definizione degl intervalli contenuto nel metodo `createEvenPattern` non fà altro che questo: calcola l'intervallo più piccolo tramite la divisione, ricava il numero degli intervalli più grandi a partire dal valore del resto e redistribuisce il resto in maniera casuale agli intervalli dei colpi che compongono il ritmo. Secondo l'esempio precedente un risultato potrebbe essere:

colpo 1 : durata intervallo = 3
colpo 2 : durata intervallo = 2

colpo 3 : durata intervallo = 3
colpo 4 : durata intervallo = 2
colpo 5 : durata intervallo = 3
colpo 6 : durata intervallo = 3

Una volta definiti gli intervalli tra un colpo e il successivo risulta semplice assegnare le posizioni dei diversi colpi. Basta assegnare il primo colpo alla posizione 1 e ricavare le posizioni successive sommando via via gli intervalli:

colpo 1 : durata intervallo = 3 ; posizione = 1 ;
colpo 2 : durata intervallo = 2 ; posizione = 1+3 = 4 ;
colpo 3 : durata intervallo = 3 ; posizione = 4+2 = 6 ;
colpo 4 : durata intervallo = 2 ; posizione = 6+3 = 9;
colpo 5 : durata intervallo = 3 ; posizione = 9+2 = 11;
colpo 6 : durata intervallo = 3 ; posizione = 11+3 = 14 ;

Come ultima operazione il metodo `createEvenPattern` passa al metodo chiamante un array in cui sono contenute le posizioni di ciascun colpo.

La definizione dei pattern dei singoli strumenti

I metodi che si occupano di disegnare i pattern dei singoli strumenti fanno sempre parte del nostro controller `GUIcontroller` e vengono tutti richiamati dal metodo `CreateEvenPattern`. Si chiamano: `designEvenKickPattern`, `designEvenSnarePattern`, `designHiHatPattern`, `designLTomPattern`, `designMTomPattern`, `designHTomPattern`. Vediamoli uno alla volta.

1) `designEvenKickPattern`

Questo metodo si occupa di creare il pattern di cassa a partire da quello massimamente distribuito e di misurarne poi la variabilità per poter assegnare un valore allo slider `variety`.

Il pattern di cassa viene definito richiamando un apposito metodo `createKickPattern`, appartenente alla classe `KickPattern`.

`createKickPattern` compie le seguenti operazioni :
- riceve l'array contenente le posizioni del pattern massimamente distribuito
- definisce un nuovo array atto a contenere i valori delle posizioni del pattern di cassa
- prende dal pattern massimamente distribuito i valori di posizione dei colpi di ordine dispari e li inserisce nell'array che ospita i valori di posizione dei colpi di cassa.

Ad esempio, se il pattern massimamente distribuito ha 5 colpi, il pattern di cassa sarà composto con le posizioni dei colpi 1, 3 e 5.

Nel caso in cui il numero di colpi del pattern massimamente distribuito sia dispari, il metodo `designEvenKickPattern` si occupa di definire, in base alla variabile casuale `randKickSnare` definita in `createEvenRhythm`, se l'ultimo colpo debba essere di cassa o rullante. Se la variabile casuale è uguale a 2 il colpo viene assegnato al rullante e l'ultimo colpo del pattern di cassa viene rimosso dall'array.

Definito il pattern di cassa `designEvenKickPattern` lo disegna sulla griglia di checkbox. L'array `kickButtons` e `kickTripletButtons` vengono prima cancellati e poi riscritti secondo gli ultimi valori di posizione contenuti nell'array `evenKickPositions`.

L'ultima operazione compiuta da `designEvenKickPattern` è definire il grado di variabilità del pattern di cassa appena definito. Questo viene fatto richiamando il metodo `getVariability` appartenente alla classe `PatternCreator`. `getVariability` recepisce un array di posizioni di colpi ed è in grado di capire se si tratta di un array di colpi di cassa o rullante. Il valore `maxShift` viene definito secondo cinque casi possibili:

a - un solo colpo in posizione uno: dato che il primo colpo è in posizione 1 il metodo riconosce il pattern come un pattern di cassa. In questo caso si tratta di un array un solo colpo quindi, dato che il primo colpo di un loop sarà sempre un colpo di cassa fisso in posizione 1, il massimo spiazzamento è 0;

b - diversi colpi con il primo in posizione uno: anche in questo caso si tratta di un pattern di cassa. Per tuttti i pattern di cassa con numero di colpi superiore a 1 il massimo spiazzamento viene definito come il numero di spazi che i colpi devono percorrere per raggrupparsi tutti

all'inizio del loop. Ad esempio, se il pattern di cassa è costituito da 4 colpi in posizione 1-5-9-13, il massimo spiazzamento sarà 18 (colpi in posizione 1-2-3-4);

c - un colpo solo, non in posizione uno: in questo caso si tratta di un colpo singolo di rullante. Il suo massimo spiazzamento è dato dal valore maggiore tra il numero di spazi da percorrere per

occupare la prima posizione del loop (`maxShiftLeft`) e il numero di spazi da percorrere per occupare l'ultima posizione del loop (`maxShiftRight`).

d - due colpi, con il primo non in posizione uno: in questo caso il calcolo è analogo a quello per la cassa, quando i colpi sono più di uno.

e - più di due colpi, con il primo non in posizione uno: si tratta di un pattern di rullante con un numero di colpi maggiore di due. Il massimo spostamento possibile è dato dalla somma degli intervalli tra un pattern e il successivo.

La variabilità del pattern di cassa è una variabile globale della classe `GUIController` e verrà quindi utilizzata nel metodo `createEvenRhythm` per drefinire la variabilità globale del pattern cassa+rullante.

2) `designEvenSnarePattern`

Il metodo `designEvenSnarePattern` crea il pattern di rullante, basandosi sempre sullo stesso pattern massimamente distribuito utilizzato per creare il pattern di cassa. Anche qui viene misurata la variabilità per poi assegnare un valore allo slider `variety`.

Nel metodo `createSNarePattern`, appartenente alla classe `SnarePattern`, si susseguono i seguenti step: .

- ricezione dell'array contenente le posizioni del pattern massimamente distribuito
- definizione di un nuovo array per contenere i valori delle posizioni del pattern di rullante
- trasferimento dei valori di posizione dei colpi di ordine pari dal pattern massimamente distribuito all'array che ospita i valori di posizione dei colpi di rullante.

Ad esempio, se il pattern massimamente distribuito ha 5 colpi, il pattern di rullante sarà composto con le posizioni dei colpi 2 e 4.

Come il metodo `designEvenKickPattern`, nel caso in cui il numero di colpi del pattern massimamente distribuito sia dispari, il metodo `designEvenSnarePattern` definisce, in base alla varaibile casuale `randKickSnare`, se l'ultimo colpo debba essere di cassa o rullante. Se la

variabile casuale è uguale a 2 l'ultimo colpo del pattern massimamente distribuito viene aggiunto all'array delle posizioni dei colpi di rullante.
Definito il pattern di rullante `designEvenSnarePattern` lo disegna sulla griglia di checkbox. L'array `snareButtons` e `snareTripletButtons` vengono cancellati e poi riscritti secondo gli ultimi valori di posizione contenuti nell'array `evenSnarePositions`.
Alla fine del metodo `designEvenKickPattern` viene definito il grado di variabilità del pattern

di rullante appena creato. Il metodo `getVariability` si occupa del calcolo secondo le modalità viste per `designEvenKickPattern` .
La variabilità del pattern di rullante è una variabile globale della classe `GUIController` e verrà poi ripresa nel metodo `createEvenRhythm` per impostare il valore di variabilità dello slider `variety`.

3) `designHiHatPattern`

Il pattern di charleston ha un comportamento più semplice dei due appena visti. Per il charleston è possibile definire solo un numero di colpi pari a 4, 8 o 16 e posizionarli solo in massima distribuzione.
La prima operazione svolta dal metodo `designHiHatPattern` è l'acquisizione del valore del numero di colpi selezionato dall'utente. Lo slider da cui tale valore verrà acquisito varierà a seconda del valore di `beatNum`, come visto precedentemente.
Infine viene creato il pattern massimamente distribuito (sempre grazie al metodo `createEvenPattern`) e vengono disegnati sulla griglia di checkbox i colpi di charleston.
Il disegno dei colpi di charleston presenta un accorgimento in più rispetto al disegno dei colpi di cassa e rullante: se il valore del numero dei colpi è elevato rispetto al numero di spazi disponibili i colpi di charleston in concomitanza a quelli di rullante vengono cancellati, in modo da emulare il movimento del batterista che suona sequenze di colpi ravvicinati su rullante e charleston. I colpi concomitanti vengono cancellati in questi casi:
- il numero di spazi del loop è pari a 16 e il numero selezionato di colpi di charleston è 16
- il numero di spazi del loop è compreso tra 8 e 16 e il numero di colpi di charleston è 8
- il numero di spazi del loop è inferiore a 8 (i colpi di charleston saranno per forza 4)

Questa cancellazione contribuisce al realismo del ritmo e conferisce un ulteriore elemento di varietà ai pattern creati.

4) `designLTomPattern, designMTomPattern, designHTomPattern`

I tre metodi di disegno del pattern dei tom si limitano a resettare i bottoni relativi ad ogni creazione di un nuovo ritmo in quanto il disegno vero e proprio dei tom è affidato al metodo `designFill`.

3.3.3 Le variazioni sul ritmo base

Una volta creato un ritmo massimamente distribuito che lo soddisfa l'utente potrà andare a modificarne la distribuzione andando ad aggiungere un certo grado di spiazzamento ai singoli colpi di cassa e rullante che lo compongono. Quando l'utente agisce sullo slider `variety` il bottone di creazione dei pattern ritmici varia il suo stato da "Create Even Rhythm" a "Create Rhythm Variation". Da ora in poi il metodo richiamato dal bottone di creazione dei ritmi non sarà più `CreateEvenRhythm`, ma `CreateRhythmVariation`.

La variazione del pattern

La variazione del pattern creato con `CreateEvenRhythm` è gestita dal metodo `CreateRhythmVariation`. Fondamentalmente questo metodo non fà altro che lanciare la creazione delle variazioni sui singoli pattern di cassa (`designKickPatternVariation`) e rullante (`designSnarePatternVariation`), sempre tenendo conto dello stato dei bottoni di Lock. Inoltre viene rigenerato il pattern di charleston (`designHihatPattern`) in modo da adattarsi ogni volta alla nuova disposizione dei colpi nel pattern di base.

Variazione dei pattern di cassa e rullante

La variazione dei pattern di cassa e rullante avviene tramite il metodo addVariation, appartenente alla classe `PatternCreator`, contenuto in `designKickPatternVariation` e `designSnarePatternVariation`. Se il valore dello slider variety è uguale a 0, il pattern di

cassa/rullante rimane quello massimamente distribuito (`evenKickPositions/ evenSnarePositions`), se invece `variety` è diverso da 0 allora viene applicato il metodo `addVariation`.
Una volta generato il pattern variato vengono ridisegnati gli array dei bottoni di cassa e rullante per rappresentare il nuovo ritmo sulla griglia di checkbox.

Metodo addVariation

Il metodo che crea le variazioni sui pattern massimamente distribuiti è chiamato `addVariation` ed appartiene alla classe `PatternCreator`. `addVariation` riceve in ingresso due parametri: il grado di varietà o spiazzamento (`aVarietyValue`), e l'array dei valori di posizione dei colpi di un pattern massimamente distribuito (`aPatternPos`).
Il valore di spiazzamento indica di quanti posti devono essere spostati i colpi del pattern fornito. Se ad esempio il grado di spiazzamento è pari a 2, e il pattern da variare è composto di 2 colpi, allora l'algoritmo per due volte sceglierà casualmente nel pattern un colpo da spostare di uno spazio. Mettiamo che il pattern in ingresso sia definito dalle posizioni 3-11. I possibili pattern risultanti da uno spiazzamento di valore 2 sono:

pattern 1: 1-11 (sposto indietro di 2 posizioni il colpo 1)
pattern 2: 5-11 (sposto avanti di 2 posizioni il colpo 1)
pattern 3: 2-10 (sposto indietro di 1 posizione il colpo 1 e il colpo 2)
pattern 4: 2-12 (sposto indietro di 1 posizione il colpo 1 e avanti di 1 posizione il colpo 2)
pattern 5: 4-10 (sposto avanti di 1 posizione il colpo 1 e indietro di 1 posizione il colpo 2)
pattern 6: 4-12 (sposto avanti di 1 posizione il colpo 1 e il colpo 2)
pattern 7: 3-9 (sposto indietro di 2 posizioni il colpo 2)
pattern 8: 3-13 (sposto avanti di 2 posizioni il colpo 2)

Se un singolo colpo viene spiazzato per più di una volta il suo allontanamento dal valore di posizione centrale massimamente distribuito aumenta. Se per esempio abbiamo un colpo che ha come posizione centrale il valore 6 e lo spiazziamo per 3 volte avremo che al primo

spiazzamento il colpo si muoverà in un range di valore 1 intorno al valore 6, al secondo spiazzamento in un range 2, al terzo in un range 3. Le posizioni possibili che può assumer saranno quindi:

spiazzamento 1: valori 5 e 7 (cioè 6-1 e 6+1)
spiazzamento 2: valori 4 e 8 (cioè 6-2 e 6+2)
spiazzamento 3: valori 3 e 9 (cioè 6-3 e 6+3)

L'algoritmo sà per quante volte deve effettuare lo spostamento di un colpo di una posizione (`varietyValue`). Ogni volta che dovrà effettuare uno spostamento, sceglierà a caso un colpo da spostare e, se possibile, lo sposterà di uno spazio in più rispetto alla sua posizione centrale che occupava nel pattern massimamente distribuito.
Ogni volta che l'algoritmo sceglie un colpo da spostare lo analizza e verifica che:
- non sia il primo colpo di un pattern di cassa, in quanto questo colpo determina l'inizio del loop e non và mai spostato.
- abbia spazio sufficiente intorno a sè per spostarsi senza uscire dal loop o oltrepassare un colpo che lo precede o lo segue.

Se il colpo selezionato è il primo colpo del pattern di cassa oppure non ha spazio sufficiente intorno per spostarsi l'algoritmo sceglie a caso un altro valore finchè non trova un colpo che è possibilie spostare.
A volte può capitare che l'algoritmo non peschi subito un valore che è possibile spostare. A questo scopo è stato inserito un valore chiamato `moved` che serve a verificare se almeno uno spostamento è stato effettuato. Quando non viene effettuato nessuno spostamento il valore `moved` viene posto a 0 e viene pescato un altro valore a caso finchè non se ne trova uno che è possibile spostare.
Quando tutti gli spostamenti sono stati effettuati `addVariation` genera un nuovo array contenente le nuove posizioni e lo passa al metodo chiamante, il quale si occuperà poi di aggiornare la griglia di checkbox con le nuove posizioni.

3.3.4 Il Player

Una volta creato un ritmo che sembra interessante per come appare sulla griglia del loop, l'utente può ascoltarlo per verificare se è di suo gradimento. L'utente può avviare o fermare la riproduzione del loop utilizzando i pulsanti play e stop, e può scegliere tra tre velocità di riproduzione: 80 BpM, 120 BpM e 140 BpM (dove BpM stà per Beats per Minute, battiti al minuto).

La riproduzione del loop

La riproduzione del loop viene avviata tramite il bottone play. Questo bottone richiama un metodo `play` che si occupa di:
1) Definire il valore dell'intervallo temporale utilizzato dal timer che regola la riproduzione temporizzata dei suoni associati ad ogni strumento.
Per motivi di sincronizzazione è stato scelto di utilizzare un singolo timer sia per scansionare i sedicesimi, che per scansionare i valori terzinati. Quindi per ottenere l'unità temporale minima ogni quarto della battuta è stato suddiviso per un valore che fosse minimo comune multiplo del numero di sedicesimi e di terzine di trentaduesimi inclusi. Essendo che in un quarto di battuta sono compresi 4 sedicesimi e 6 terzine di trentaduesimi, il valore per cui dividere l'unità temporale del quarto è risultato essere 12. In questo modo ogni quarto è suddiviso in 12 intervalli temporali. Se associassimo un suono ad ogni checkbox che stà nell'intervallo di un quarto otterremmo il seguente risultato:

- istante 1 : suonano sedicesimo #1 + terzina #1
- istante 2 : non suona nulla
- istante 3 : suona la terzina #2
- istante 4 : suona il sedicesimo #2
- istante 5 : suona la terzina #3
- istante 6 : non suona nulla
- istante 7 : suonano sedicesimo #3 + terzina #4
- istante 8 : non suona nulla

- istante 9 : suona terzina #5
- istante 10 : suona sedicesimo #4
- istante 11: suona terzina #6
- istante 12 : non suona nulla

Grazie a questo valore di scansione temporale è possibile suonare in sequenza sia i campioni audio assegnati ai sedicesimi, che quelli assegnati alle terzine.

2) Allocare i campioni audio

Tutti i campioni audio utilizzati dall'applicazione vengono allocati al momento in cui viene schiacciato play. Allocare la memoria per i campioni all'esterno del metodo richiamato dal timer rende più semplice all'applicazione riprodurre i campioni rispettandola temporizzazione.

Per riprodurre in maniera più realistica un ritmo suonato con una batteria acustica ad ogni strumento sono stati associati più campioni. Questo permette ad ogni strumento di non avere sempre lo stesso timbro ogni volta che viene suonato e quindi di simulare meglio un ritmo reale suonato da un essere umano con una batteria acustica.

3) avviare il timer

Il timer una volta avviato richiama ad ogni intervallo di tempo `timeInt` il metodo `playSound`. L'intervallo di tempo `timeInt` ha il valore di un dodicesimo di `(60/bpmValue)`, per i motivi spiegati in precedenza. Una volta avviato, il timer non si ferma finchè non viene premuto il tasto stop.

4) abilitare il tasto stop e disabilitare il tasto play

Questa operazione impedisce all'utente di avviare per due volte consecutive la riproduzione del loop senza prima averla fermata tramite il pulsante stop.

Il metodo playSound

Il metodo `playSound` si occupa di visualizzare lo scorrere del tempo sulla griglia di checkbox e suonare i campioni quando trova le checkbox in stato "acceso".
Lo scorrere del tempo viene visualizzato tramite il cambiamento dello stato delle checkbox. Le checkbox associate al valore temporale corrente del timer cambiano di colore diventando più chiare, per poi essere ripristinate al loro stato originario non appena il valore temporale del timer cambia.
I valori di `timePos`, `timePosBin` e `timePosTer` vengono utilizzati per definire le posizioni temporali relative del timer, delle checkbox che rappresentano i sedicesimi e delle checbox che rappresentano le terzine. Essi sono incrementati ad ogni chiamata del metodo `playSound`.
Il primo `if` all'interno di `playSound` definisce la struttura del loop. Ogni volta che il timer oltrepassa la fine della battuta (`timePos == ((beatNumValue*3)+1)` i valori di `timePos`, `timePosBin` e `timePosTer` vengono resettati, così da poter ricominciare la scansione della battuta in loop.
Il secondo e il terzo `if` servono invece a verificare se il valore corrente di `timePos` è uguale al valore di posizione temporale di un sedicesimo (`(((timePos+2)%3) == 0)`) o di una terzina (`(((timePos+1)%2) == 0)`.Nel caso la condizione sia verificata le checkbox associate vengono cambiate di stato e le checkbox dell'istante temporale precedente vengono ripristinate al loro stato originale.
Infine giungiamo alla riproduzione vera e propria dei campioni. Per ogni strumento viene verificato se la checkbox associata è in stato "acceso" e se il corrispettivo tasto Mute è premuto. Nel caso la checkbox sia attiva e il tasto Mute non sia premuto viene scelto a caso uno dei campioni disponibili per quello strumento e il suo suono viene riprodotto. I diversi campioni presi per un singolo strumento hanno valori diversi di velocity, ovvero il loro timbro varia in relazione all'intensità con cui quel campione è stato eseguito. In questo modo le riproduzioni consecutive di uno stesso strumento risultano meno innaturali e più verosimili.

Per terminare la riproduzione sonora del ritmo l'utente preme il tasto stop. Il tasto stop richiama il metodo omonimo, il quale compie le seguenti operazioni:

- ripristino dello stato dell'ultima checkbox scansionata dal loop,
- disabilitazione del tasto stop (in modo che l'utente non lo schiacci per due volte consecutive senza prima aver premuto play)
- abilitazione del tasto play
- deallocazione di tutti i campioni

Modifica del loop in tempo reale

La struttura di play dei campioni basata sullo stato delle checkbox permette di ricavare dal programma una utile funzionalità.
L'utente, durante la riproduzione sonora del loop generato in maniera automatica, può variarne in tempo reale la struttura aggiungendo e togliendo colpi tramite l'accensione o lo spegnimento delle checkbox. Questo permette di sperimentare velocemente diverse soluzioni ritmiche e rende il programma uno strumento per la creazione di ritmi flessibile e potente.

3.3.5 Il generatore di fill

I fill solitamente vengono utilizzati all'interno delle strutture ritmiche dei brani musicali per rivitalizzare un ritmo che si ripete in maniera uguale da troppo tempo o per anticipare le variazioni strutturali del brano stesso (passaggio da prima strofa a seconda strofa, da strofa a ritornello, etc...).
Attraverso il generatore di fill l'utente può creare variazioni alle parti finali dei ritmi appena creati e utilizzarle per comporre le strutture dei suoi pezzi. Tramite l'apposito silder è possibile creare fill che coprano l'ultimo quarto della battuta (fill da un quarto), la seconda metà (fill da due quarti) o l'intera battuta (fill da quattro quarti).

La creazione dei pattern dei fill

Per la creazione automatica dei pattern di fill sono stati analizzati fill reali al fine di estrarne le caratteristiche costituive.
Ognuno dei fill analizzati è stato suddiviso in sottounità strutturali della lunghezza di un ottavo, chiamati Chunk. In questo modo sono stati definiti 5 chunk fondamentali, ognuno dei quali rappresenta una possibile sequenza di colpi all'interno di un ottavo:

XX = due colpi consecutivi suonati sui sedicesimi
XO = un colpo suonato sul primo sedicesimo, seguito da silenzio
OX = un colpo suonato sul secondo sedicesimo, preceduto da silenzio

XXX = tre colpi consecutivi suonati sulle terzine di trentaduesimi
OO = nessun colpo viene suonato

Nella tabella successiva è possibile osservare la struttura di uno dei fill analizzati (durata = 2 quarti) e come sono stati ricavate le strutture dei chunk. Come si può vedere il fill presenta un chunk XX all'ottavo #1, una terzina XXX all'ottavo #2, un secondo chunk XX all'ottavo #3, ed infine un chunk XO all'ottavo #4.

	#1		#2			#3		#4	
Kick	X		X						
Snare		X				X	X		
HiHat									
HiTom				X				X	
MidTom					X				
LowTom									
	XX		XXX			XX		XO	

Ogni chunk può presentare una differente combinazione strumentale. Ad esempio un chunk XX può essere composto da: Snare+Snare, Kick+Kick, Kick+Snare, etc... Per ogni chunk fondamentale sono state individuate le possibili combinazioni timbriche e calcolate le ricorrenze di tali combinazioni in ogni ottavo di cui è composto il fill. Di seguito un esempio di tabella delle ricorrenze delle combinazioni timbriche relativa all'analisi del chunk XX nei fill da un quarto.

	#1	#2	#1 %	#2 %
Snare+Snare (SS)	6	4	55	50
Snare+Kick (SK)	1	1	9	13
Snare+Tom (ST)	2		18	
Kick+Kick (KK)		1		13
Kick+Snare (KS)		1		12
Kick+Tom (KT)	1		9	
Tom+Tom - uguali (TTu)	1		9	
Tom+Tom - diversi (TTd)		1		12

La tabella mostra il numero di ricorrenze in termini assoluti e in termini percentuali. Da questa si possono ricavare informazioni interessanti sulla composizione dei fill, ad esempio:

- la combinazione strumentale che ricorre maggiormente è SS, mentre le altre combinazioni ricorrono in maniera equivalente
- il pattern XX ricorre maggiormente sul primo chunk rispetto al secondo

- le combinazioni KK, KS e TTd non vengono mai suonate sul primo chunk
- le combinazioni ST, KT, TTu non vengono mai suonate sul secondo chunk
- etc...

Da tabelle come la precedente è quindi possibile ricavare una distribuzione statistica dei diversi chunk fondamentali e delle diverse combinazioni strumentali. “Insegnare” queste distribuzioni statistiche all’algoritmo che calcola il fill pattern è un modo in cui possiamo ottenere fill realistici in maniera automatica.

I metodi che si occupano della creazione dei fill pattern sono tre: `createFill`, `composeFill` e `createFillChunk`.

1) `createFill`

Il metodo `createFill` appartiene alla classe `GUIController`. Esso richiama il metodo `composeFill` passandogli il numero di chunk da comporre in base al valore di lunghezza del fill selezionato dall’utente. Una volta creato il fill lo passa al metodo `designFill` per la rappresentazione sulla griglia delle checkbox.

2) `composeFill`

Il metodo `composeFill` appartiene invece alla classe `FillCreator`. `composeFill` genera:

- l’array `fillPos` che conterrà la struttura del fill
- l’array `chunk`, con cui `fillPos` verrà riempito man mano che ogni chunk del fill viene costruito.
- i valori degli oggetti `NSNumber` sonservono ad inserire nel chunk l’indicazione su quali strumenti contiene e sulla sua natura ternaria o binaria.

`composeFill` infine richiama `createFillChunk` per geneare i chunk con cui comporre l’array `fillPos`, passandogli l’informazione su che tipo di chunk deve comporre (lunghezza del fill e posizione del chunk all’interno del fill). Una volta che è stato composto `fillPos` viene passato al metodo chiamante, `createFill`.

3) `createFillChunk`

`createFillChunk` riceve in ingresso le informazioni relative a quale chunk si vuole che venga costruito. Grazie a queste informazioni (lunghezza del fill e posizione del chunk all’interno del fill) il metodo crea il chunk basandosi sulla distribuzione statistica di chunk fondamentali e combinazioni strumentali tipiche del chunk stesso. Una volta individuata la distribuzione

statistica corretta `createFillChunk` genera 2 valori casuali i cui valori sono compresi tra 1 e 100: il primo viene utilizzato per scegliere il tipo di chunk fondamentale, il secondo per scegliere il tipo di combinazione strumentale.

Facciamo un esempio:

Il metodo riceve la richiesta di generare il primo chunk di un fill lungo un quarto. La sua distribuzione di probabilità dell'esistenza dei chunk fondamentali è:

```
XX = 73 :     73 volte su 100 il chunk è di tipo XX
XO = 0 :      questo tipo di chunk non viene suonato in questa posizione
OX = 0 :      questo tipo di chunk non viene suonato in questa posizione
XXX = 27 :    27 volte su 100 il chunk è di tipo XXX
OO = 0 :      questo tipo di chunk non viene suonato in questa posizione
```

Il range percentuale del chunk XX viene impostato come [1..73], mentre quello del chunk XXX risulta essere [74..100].

Poniamo che il metodo generi un primo valore casuale pari a 58. Il chunk sarà di tipo XX.

Definiamo ora la combinazione strumentale. La distribuzione di probabilità di esistenza delle combinazioni strumentali di un chunk XX posizionato al primo posto di un fill di lunghezza un quarto è:

```
SS = 55 :     55 volte su 100 la comb. strum. è data da Snare+Snare
SK = 9 :      9 volte su 100 la comb. strum. è data da Snare+Kick
ST = 18 :     18 volte su 100 la comb. strum. è data da Snare+Tom
KK = 0 :      questo tipo di comb. strum. non viene suonata in questa posizione
KS = 0 :      questo tipo di comb. strum. non viene suonata in questa posizione
KHh = 0 :     questo tipo di comb. strum. non viene suonata in questa posizione
KT = 9 :      9 volte su 100 la comb. strum. è data da Kick+Tom
TT = 9 :      9 volte su 100 la comb. strum. è data da Tom+Tom (uguali)
TTd = 0 :     questo tipo di comb. strum. non viene suonata in questa posizione
TK = 0 :      questo tipo di comb. strum. non viene suonata in questa posizione
TS = 0 :      questo tipo di comb. strum. non viene suonata in questa posizione
HhHh = 0 :    questo tipo di comb. strum. non viene suonata in questa posizione
HhS = 0 :     questo tipo di comb. strum. non viene suonata in questa posizione
HhK = 0 :     questo tipo di comb. strum. non viene suonata in questa posizione
HhT = 0 :     questo tipo di comb. strum. non viene suonata in questa posizione
```

I range percentuali vengono assegnati in questo modo:

SS: [1..55]
SK: [56..64]
ST: [65..82]
KT: [83..91]
TT: [92..100]

Poniamo che il metodo generi un secondo valore casuale pari a 72. Il chunk XX sarà costituito da un colpo di rullante in prima posizione e un colpo di tom in seconda.

Abbiamo visto come viene costruito un chunk ma non come si distingue un chunk ternario da uno binario. La distinzione avviene in questo modo. Ogni chunk, sia ternario che binario, è un array costituito da tre elementi. Il primo elemento di un chunk distingue tra binario e ternario. Se il primo elemento è l'oggetto "b" (binario) significa che il chunk è binario, se invece è diverso da "b", conterrà un oggetto che rappresenta uno degli strumenti della batteria, quindi significa che l'array è ternario. Il secondo e il terzo elemento di tutti i chunk invece contengono sempre o l'elemento "v" (vuoto = non suona alcun strumento) o uno strumento.

Costruito il chunk, il metodo `createFillChunk` lo passa al metodo `composeFill` che lo usa per costruire il pattern di fill.

La rappresentazione dei pattern dei fill

I pattern di fill vengono rappresentati sulla griglia di checkbox attraverso il metodo `designFill`, il quale a sua volta disegna i chunk binari e ternari richiamando i metodi `designBinChunk` e `designTerChunk`.

1) `designFill`

Il metodo `designFill` è suddiviso in tre sezioni, ognuna delle quali si occupa di uno dei seguenti casi: fill di lunghezza 1 quarto, fill di lunghezza 2 quarti, fill di lunghezza 4 quarti. Ognuna delle tre sezioni compie le stesse due operazioni: spegnere tutte le checkbox nello spazio in cui deve essere creato il fill e disegnare il fill pattern chunk per chunk richiamando i metodi di disegno dei chunk. Ogni volta che designFill chiede di disegnare un chunk sulla griglia, passa ai metodi preposti la posizione del chunk sulla griglia e i valori che rappresentano gli strumenti da disegnare.

2) `designBinChunk` e `designTerChunk`

Questi due metodi non fanno altro che puntare alla posizione della griglia passata da `designFill` e attivare le checkbox corrispondenti agli strumenti codificati nei parametri `aVal1`, `aVal2`, `aVal3`.

Queste le codifiche dei diversi strumenti:

val = 0 : checkbox vuota
val = 1: cassa
val = 2: rullante
val = 3: charleston
val = 4: timpano
val = 5: tom medio
val = 6: tom piccolo

4

CONCLUSIONI

Il programma di autocomposizione presentato contiene al suo interno la caratteristica fondamentale per la generazione dei pattern ritmici: la disposizione dei colpi in posizione massimamente distribuita. La distribuzione massima dei colpi (cioè a distanze intervallari regolari) permette di ottenere ritmi verosimili, sicuramente utili alla composizione di brani musicali o ad essere utilizzati come base di accompagnamento per lo studio di altri strumenti. Tuttavia l'utilizzo esclusivo di questa tecnica porta a privilegiare solo l'aspetto "distensivo" di un ritmo, cioè quello della regolarità, trascurando in toto quel "movimento" e quella varietà che un ritmo deve possedere per risultare accattivante e coinvolgente.

Per questo motivo è stata introdotta la possibilità per l'utente di aggiungere un grado di variabilità in seguito alla generazione del pattern a distribuzione massima. Questa variabilità è in sostanza la somma delle distanze dei singoli colpi dalla posizione di massima distribuzione. Maggiore sarà questa distanza, maggiore sarà la differenza tra il ritmo variato e il ritmo "padre" da cui deriva, e maggiore sarà il grado di movimento introdotto nella sequenza intervallare.

A queste caratteristiche il programma aggiunge la possibilità di creare strutture ritmiche per la composizione di brani musicali, grazie alla funzione di generazione di fill, quelle variazioni a fine battuta usate dai batteristi per evidenziare la struttura compositiva del brano.

Con queste funzioni il programma proposto in questo lavoro si rivela un buono strumento di ispirazione compositiva per chi cerca suggerimenti ritmici per le proprie creazioni musicali; l'applicazione inoltre si dimostra interessante per quanto riguarda eventuali integrazioni delle sue funzionalità.

Ad esempio lo spiazzamento ad una certa distanza dalla posizione centrale di massimo equilibrio svolto in maniera casuale non garantisce molto sulla qualità del pattern generato. Ritmi con pari

distanza di spiazzamento possono avere per un ascoltatore un grado di piacevolezza o di coinvolgimento molto diverso.

A fronte di ciò, una futura funzionalità che si potrebbe introdurre nell'applicazione potrebbe essere la valutazione del grado di piacevolezza dei pattern creati, fornendola come indicazione all'utente. Il grado di piacevolezza può essere valutato tramite algoritmi che calcolino alcuni dei parametri esaminati nel secondo capitolo del presente lavoro, come: l'andamento dello spettro degli intervalli (par. 2.2.3), la lunghezza del massimo intervallo di silenzio (par. 2.2.3), l'omometricità e la complementarità di due pattern combinati (par. 2.2.3), il risultato del confronto dei valori di distanza (par. 2.2.4) o di complessità (par. 2.2.6) ottenuti tramite tecniche diverse. Anche altri parametri non contenuti in questa trattazione possono risultare utili, come la rythmic oddity o la off-beatness, che misurano rispettivamente il grado di simmetria geometrica e il grado di corrispondenza dei colpi in battuta a distribuzioni canoniche.

Un altro sviluppo funzionale potrebbe essere l'introduzione del genere musicale come parametro di scelta dell'utente. In questo caso si dovrebbe avviare uno studio approfondito che definisca prima di tutto le regole di costruzione dei pattern a seconda dei diversi generi musicali, e in secondo luogo gli insiemi strumentali che caratterizzano ogni genere, essendo i timbri parte integrante di uno stile musicale.

Sicuramente poi sarebbe utile introdurre la visualizzazione geometrica dei pattern costruiti, in modo tale da fornire all'utente un feedback visivo potente della struttura del ritmo, magari evidenziando le caratteristiche geometriche salienti come la presenza di triangoli isosceli o ad angolo retto all'interno del poligono ritmico.

Un programma con tali caratteristiche sarebbe sicuramente un valido strumento per la composizione automatica di parti ritmiche di brani musicali. Gli algoritmi sviluppati potrebbero inoltre rivelarsi utili anche nell'ambito frequenziale per la composizione di strutture armoniche e melodiche.

APPENDICE: IL CODICE del PROGRAMMA

A.1 L'impostazione iniziale dell'interfaccia

================ GUIController - awakeFromNib ================

```
- (void)awakeFromNib{

// Disabilitazione delle parti di GUI non disponibili all'apertura
[variety setEnabled:NO];  [variety setNumberOfTickMarks:2];
[kickLock setEnabled:NO];  [snareLock setEnabled:NO];  [hihatLock setEnabled:NO];
[lTomLock setEnabled:NO];  [mTomLock setEnabled:NO];  [hTomLock setEnabled:NO];
[playButton setEnabled:NO];  [stopButton setEnabled:NO];  [bpm setEnabled:NO];
[hh48 setHidden:YES];  [hh4 setHidden:YES];
[fillLength setEnabled:NO];  [generateFill setEnabled:NO];
[createRtmVariation setHidden:YES];

// Allocazione degli array di bottoni di: Kick, Snare, Hihat, LTom, MTom, HTom
k00 = [[NSButton alloc]init];
kickButtons = [[NSArray alloc]
initWithObjects:k00,k01,k02,k03,k04,k05,k06,k07,k08,k09,k10,k11,k12,k13,k14,k15,k16,ni
l];
kt00 = [[NSButton alloc]init];
kickTripletButtons = [[NSArray alloc]
initWithObjects:kt00,kt01,kt02,kt03,kt04,kt05,kt06,kt07,kt08,kt09,kt10,kt11,kt12,kt13,
kt14,kt15,kt16,kt17,kt18,kt19,kt20,kt21,kt22,kt23,kt24,nil];

sn00 = [[NSButton alloc]init];
snareButtons = [[NSArray alloc]
initWithObjects:sn00,sn01,sn02,sn03,sn04,sn05,sn06,sn07,sn08,sn09,sn10,sn11,sn12,sn13,
sn14,sn15,sn16,nil];
snt00 = [[NSButton alloc]init];
snareTripletButtons = [[NSArray alloc]
initWithObjects:snt00,snt01,snt02,snt03,snt04,snt05,snt06,snt07,snt08,snt09,snt10,snt1
1,snt12,snt13,snt14,snt15,snt16,snt17,snt18,snt19,snt20,snt21,snt22,snt23,snt24,nil];

hh00 = [[NSButton alloc]init];
hihatButtons = [[NSArray alloc]
initWithObjects:hh00,hh01,hh02,hh03,hh04,hh05,hh06,hh07,hh08,hh09,hh10,hh11,hh12,hh13,
hh14,hh15,hh16,nil];
hht00 = [[NSButton alloc]init];
hihatTripletButtons = [[NSArray alloc]
initWithObjects:hht00,hht01,hht02,hht03,hht04,hht05,hht06,hht07,hht08,hht09,hht10,hht1
1,hht12,hht13,hht14,hht15,hht16,hht17,hht18,hht19,hht20,hht21,hht22,hht23,hht24,nil];

lt00 = [[NSButton alloc]init];


lTomButtons = [[NSArray alloc]
initWithObjects:lt00,lt01,lt02,lt03,lt04,lt05,lt06,lt07,lt08,lt09,lt10,lt11,lt12,lt13,
lt14,lt15,lt16,nil];
ltt00 = [[NSButton alloc]init];
lTomTripletButtons = [[NSArray alloc]
initWithObjects:ltt00,ltt01,ltt02,ltt03,ltt04,ltt05,ltt06,ltt07,ltt08,ltt09,ltt10,ltt1
```

```
1,ltt12,ltt13,ltt14,ltt15,ltt16,ltt17,ltt18,ltt19,ltt20,ltt21,ltt22,ltt23,ltt24,nil];

mt00 = [[NSButton alloc]init];
mTomButtons = [[NSArray alloc]
initWithObjects:mt00,mt01,mt02,mt03,mt04,mt05,mt06,mt07,mt08,mt09,mt10,mt11,mt12,mt13,
mt14,mt15,mt16,nil];
mtt00 = [[NSButton alloc]init];
mTomTripletButtons = [[NSArray alloc]
initWithObjects:mtt00,mtt01,mtt02,mtt03,mtt04,mtt05,mtt06,mtt07,mtt08,mtt09,mtt10,mtt1
1,mtt12,mtt13,mtt14,mtt15,mtt16,mtt17,mtt18,mtt19,mtt20,mtt21,mtt22,mtt23,mtt24,nil];

ht00 = [[NSButton alloc]init];
hTomButtons = [[NSArray alloc]
initWithObjects:ht00,ht01,ht02,ht03,ht04,ht05,ht06,ht07,ht08,ht09,ht10,ht11,ht12,ht13,
ht14,ht15,ht16,nil];
htt00 = [[NSButton alloc]init];
hTomTripletButtons = [[NSArray alloc]
initWithObjects:htt00,htt01,htt02,htt03,htt04,htt05,htt06,htt07,htt08,htt09,htt10,htt1
1,htt12,htt13,htt14,htt15,htt16,htt17,htt18,htt19,htt20,htt21,htt22,htt23,htt24,nil];

// Nessun ritmo ancora creato
rhythmCreated = NO;
}
```

A.2 La creazione del ritmo

================ GUIController - beatChange ================

```
- (IBAction)beatChange:(id)sender{
[variety setEnabled:NO];
```

```
// controllo del valore dello slider Density
if ([beatNum intValue] < 8){
     [hitNum setMaxValue: [beatNum intValue]];
     [hitNum setNumberOfTickMarks:[beatNum intValue]];}
if ([beatNum intValue] >= 8){
     [hitNum setMaxValue:8];
     [hitNum setNumberOfTickMarks:8];}

// controllo del valore dello slider dell'hi-hat
if ([beatNum intValue] == 16){
     [hh4 setHidden:YES];  [hh4816 setHidden:NO];  [label16 setHidden:NO];
     [hh48 setHidden:YES];  [label4 setHidden:NO];  [label8 setHidden:NO];}
if (([beatNum intValue] < 16) && ([beatNum intValue] >= 8)){
     [hh4 setHidden:YES];  [hh4816 setHidden:YES];  [label16 setHidden:YES];
     [hh48 setHidden:NO];  [label4 setHidden:NO];  [label8 setHidden:NO];}


if (([beatNum intValue] < 8) && ([beatNum intValue] >= 4)){
     [hh4816 setHidden:YES];  [label16 setHidden:YES];  [label8 setHidden:YES];
     [hh48 setHidden:YES];  [label4 setHidden:NO];  [hh4 setHidden:NO];}
if ([beatNum intValue] < 4){
     [hh4 setHidden:YES];  [hh4816 setHidden:YES];  [hh48 setHidden:YES];
     [label16 setHidden:YES];  [label8 setHidden:YES];  [label4 setHidden:YES];}

//attivazione del numero di switch pari a beatNum
for (n = 1; n <= 16 ; n++){
     if (n <= [beatNum intValue]){
          [[kickButtons objectAtIndex:n] setTransparent:NO];
          [[snareButtons objectAtIndex:n] setTransparent:NO];
          [[hihatButtons objectAtIndex:n] setTransparent:NO];
          [[lTomButtons objectAtIndex:n] setTransparent:NO];
          [[mTomButtons objectAtIndex:n] setTransparent:NO];
          [[hTomButtons objectAtIndex:n] setTransparent:NO];}
     else {
          [[kickButtons objectAtIndex:n] setTransparent:YES];
          [[snareButtons objectAtIndex:n] setTransparent:YES];
          [[hihatButtons objectAtIndex:n] setTransparent:YES];
          [[lTomButtons objectAtIndex:n] setTransparent:YES];
          [[mTomButtons objectAtIndex:n] setTransparent:YES];
          [[hTomButtons objectAtIndex:n] setTransparent:YES];}}
for (n = 1; n <= 24 ; n++){
     if (n <= (([beatNum intValue]*3)/2)){
          [[kickTripletButtons objectAtIndex:n]setTransparent:NO];
          [[snareTripletButtons objectAtIndex:n] setTransparent:NO];
          [[hihatTripletButtons objectAtIndex:n] setTransparent:NO];


          [[lTomTripletButtons objectAtIndex:n] setTransparent:NO];
          [[mTomTripletButtons objectAtIndex:n] setTransparent:NO];
          [[hTomTripletButtons objectAtIndex:n] setTransparent:NO];}
     else{
          [[kickTripletButtons objectAtIndex:n] setTransparent:YES];
          [[snareTripletButtons objectAtIndex:n] setTransparent:YES];
          [[hihatTripletButtons objectAtIndex:n] setTransparent:YES];
          [[lTomTripletButtons objectAtIndex:n] setTransparent:YES];
          [[mTomTripletButtons objectAtIndex:n] setTransparent:YES];
          [[hTomTripletButtons objectAtIndex:n] setTransparent:YES];}}

// crea automaticamente di ritmo solo se il bottone createEvenRhythm è stato già
schiacciato
if (rhythmCreated == YES){
     [self createEvenRhythm:self];}
}
```

========== GUIController - densityChange ==========

```
- (IBAction)densityChange:(id)sender{
[variety setIntValue:0];
[variety setEnabled:NO];

[createRtmVariation setHidden:YES];
[createEvenRtm setHidden:NO];
}
```

========== GUIController - fourEightSixteenChange ==========

```
- (IBAction)fourEightSixteenChange:(id)sender{
if ([hihatLock state] == NSOffState){
     [self designHihatPattern];}
}
```

========== GUIController - fourEightChange ==========

```
- (IBAction)fourEightChange:(id)sender{
if ([hihatLock state] == NSOffState){
     [self designHihatPattern];}
}
```

========== GUIController - createEvenRhythm ==========

```
- (IBAction)createEvenRhythm:(id)sender{

// impostazione della GUI
if (![variety isEnabled]){
     [variety setEnabled:YES];}
[kickLock setEnabled:YES];  [snareLock setEnabled:YES];  [hihatLock setEnabled:YES];
[lTomLock setEnabled:YES];  [mTomLock setEnabled:YES];  [hTomLock setEnabled:YES];
[fillLength setEnabled:YES];  [generateFill setEnabled:YES];


if (![stopButton isEnabled]){
     [playButton setEnabled:YES]; [bpm setEnabled:YES];}

// valori degli slider
beatNumValue = [beatNum intValue];
hitNumValue = [hitNum intValue];
varietyValue = [variety intValue];

// un ritmo è stato creato
rhythmCreated = YES;

// creazione del pattern con massima distribuzione
evenHitPositions = [patternCreator createEvenPattern:beatNumValue
withHits:hitNumValue];

// generazione di un numero casuale per l'assegnazione dell'ultimo colpo quando il
numero di colpi totale è dispari
randKickSnare = 0;
if ((hitNumValue % 2) != 0){
     randKickSnare = (rand() % 2)+1;}

// creazione degli EvenPattern in base allo stato dei Lock
if ([kickLock state] == NSOffState){
     [self designEvenKickPattern];}
if ([snareLock state] == NSOffState){
     [self designEvenSnarePattern];}
if ([hihatLock state] == NSOffState){
     [self designHihatPattern];}
if ([lTomLock state] == NSOffState){
     [self designLTomPattern];}
```

```
if ([mTomLock state] == NSOffState){
     [self designMTomPattern];}
if ([hTomLock state] == NSOffState){
     [self designHTomPattern];}

// calcola il massimo shift ammesso per il pattern cassa+rullante creato
if ((maxKickShift <= maxSnareShift) && (hitNumValue != 2) && (randKickSnare != 2)){
          maxShift = maxKickShift;}
else {
     maxShift = maxSnareShift;}
if (hitNumValue == 1){
     maxShift == 0;}

// imposto slider variety
[variety setMaxValue: maxShift];
[variety setNumberOfTickMarks:(maxShift+1)];

}
```

========== PatternCreator - createEvenPattern ==========

```
- (NSMutableArray*)createEvenPattern:(int)aBeatNumValue withHits:(int)aHitNumValue{

beatNumValue = aBeatNumValue;
hitNumValue = aHitNumValue;


// array di 16 colpi inizializzato
evenHit0 = [[Hit alloc]init];
evenHit1 = [[Hit alloc]init];  evenHit2 = [[Hit alloc]init];
evenHit3 = [[Hit alloc]init];  evenHit4 = [[Hit alloc]init];
evenHit5 = [[Hit alloc]init];  evenHit6 = [[Hit alloc]init];
evenHit7 = [[Hit alloc]init];  evenHit8 = [[Hit alloc]init];
evenHit9 = [[Hit alloc]init];  evenHit10 = [[Hit alloc]init];
evenHit11 = [[Hit alloc]init];  evenHit12 = [[Hit alloc]init];
evenHit13 = [[Hit alloc]init];  evenHit14 = [[Hit alloc]init];
evenHit15 = [[Hit alloc]init];  evenHit16 = [[Hit alloc]init];
evenHitEnd = [[Hit alloc]init];

evenHits = [[NSArray alloc]
initWithObjects:evenHit0,evenHit1,evenHit2,evenHit3,evenHit4,evenHit5,evenHit6,evenHit
7,evenHit8,evenHit9,evenHit10,evenHit11,evenHit12,evenHit13,evenHit14,evenHit15,evenHi
t16,evenHitEnd,nil];

// Assegnazione dell'ordine degli Hits e delle posizioni iniziali e finali
for ( n = 1; n <= hitNumValue ; n++ ){
     [[evenHits objectAtIndex:n] setOrder:n];}
     [evenHit0 setHitPosition:0];                //posizione iniziale
     [[evenHits objectAtIndex:(hitNumValue+1)] setHitPosition:(beatNumValue+1)];

//Definizione degli INTERVALLI - REGOLA di DISTRIBUZIONE MAX
int intNum = hitNumValue;
int rest = beatNumValue % hitNumValue;
int intNumPlusOne = rest;                              // numero di intervalli
maggiorati di uno
int intNumPlusZero = intNum-intNumPlusOne;   // numero di intervalli non maggiorati

for ( n = 1; n <= intNum ; n++ ) {
     int randval = rand() % 2;

     int intDur = [[evenHits objectAtIndex:n] getIntervalDuration]; //variabile
d'appoggio

     if ((intNumPlusOne != 0) && (intNumPlusZero != 0)){
          switch (randval) {
               case 0:
                    intDur = (beatNumValue / hitNumValue);
```

```
                    intNumPlusZero = intNumPlusZero-1;break;
               case 1:
                    intDur = (beatNumValue / hitNumValue)+1;
                    intNumPlusOne = intNumPlusOne-1;break;}}
     else if (intNumPlusOne == 0){
          intDur = (beatNumValue / hitNumValue);
          intNumPlusZero = intNumPlusZero-1;}
     else {                                  // (intNumPlusZero == 0)
          intDur = (beatNumValue / hitNumValue)+1;
          intNumPlusOne = intNumPlusOne-1;}

     [[evenHits objectAtIndex:n] setIntervalDuration:intDur];
}
// Definizione delle POSIZIONI in base agli intervalli
//[[evenHits objectAtIndex:1] setCenter:1];
[[evenHits objectAtIndex:1] setHitPosition:1];


for ( n = 2; n <= hitNumValue ; n++ ) { // la posizione del colpo 1 è già definita
     prevPos = [[evenHits objectAtIndex:(n-1)] getHitPosition];         //var
d'appoggio
     int intDur = [[evenHits objectAtIndex:(n-1)] getIntervalDuration]; //var
d'appoggio

     [[evenHits objectAtIndex:n] setHitPosition:(prevPos + intDur)];}

// Crea Array da passare al GUIController
positions = [[NSMutableArray alloc]init];
pos0 = [[NSNumber alloc]initWithInt:0];
[positions addObject:pos0];
int numHit = 1;

for (n = 1; n <= beatNumValue ; n++){
     if ((([[evenHits objectAtIndex:numHit] getHitPosition]) == n){
          [positions addObject:[[NSNumber alloc]initWithInt:n]];
          numHit++;}}

return positions;
}
```

============ PatternCreator - getVariability ============

```
- (int)getVariability:(NSMutableArray*)aPatternPos withBeatNum:(int)aBeatNum{

patternPos = aPatternPos;
beatNumValue = aBeatNum;

maxShift = 0;

// 1 colpo di kick
if ((([patternPos objectAtIndex:1] intValue] == 1) && ([patternPos count] == 2)){
     maxShift = 0;};
// più di un colpo di kick
if ((([patternPos objectAtIndex:1] intValue] == 1) && ([patternPos count] != 2)){
     for (n = 1; n <= ([patternPos count]-1) ; n++){
          maxShift = maxShift + (([patternPos objectAtIndex:n] intValue] - n);}}
// 1 colpo di snare
if ((([patternPos objectAtIndex:1] intValue] != 1) && ([patternPos count] == 2)){
     int maxShiftLeft = (([patternPos objectAtIndex:1]intValue]-1);
     int maxShiftRight = (beatNumValue - (([patternPos objectAtIndex:1]intValue]-1))-1;
     if (maxShiftLeft >= maxShiftRight){
          maxShift = maxShiftLeft;}
     else {
          maxShift = maxShiftRight;}}
// 2 colpi di snare
if ((([patternPos objectAtIndex:1] intValue] != 1) && ([patternPos count] == 3)){
     for (n = 1; n <= ([patternPos count]-1) ; n++){
          maxShift = maxShift + ((([patternPos objectAtIndex:n] intValue] -
```

```
[[patternPos objectAtIndex:n-1] intValue])-1);}}

// più di 2 colpi di snare
if ((([patternPos objectAtIndex:1] intValue] != 1) && ([patternPos count] > 3)){
     for (n = 2; n <= ([patternPos count]-1) ; n++){
          maxShift = maxShift + ((([patternPos objectAtIndex:n] intValue] -

[[patternPos objectAtIndex:n-1] intValue])-1);}}

return maxShift;
}
```

============ GUIController - designEvenKickPattern ============

```
- (void)designEvenKickPattern{

// creo il kickpattern massimamente distribuito
evenKickPositions = [kickPattern createKickPattern:evenHitPositions];

// assegnazione dell'ultimo colpo quando il numero totale di colpi è dispari
if ((randKickSnare == 2) && (hitNumValue > 2)){
     [evenKickPositions removeLastObject];}

// restore dello stato iniziale degli switch (tutti OFF)
for (n = 1; n <= beatNumValue ; n++){
     [[kickButtons objectAtIndex:n] setState:NSOffState];}
for (n = 1; n <= ((beatNumValue*3)/2) ; n++){
     [[kickTripletButtons objectAtIndex:n] setState:NSOffState];}

// disegna i kickSwitches
for (n = 1; n <= ([evenKickPositions count]-1) ; n++){
     int position = [[evenKickPositions objectAtIndex:n] intValue];
     [[kickButtons objectAtIndex:position] setState:NSOnState];}

// misuro la variabilità del even Kick pattern
maxKickShift = [patternCreator getVariability:evenKickPositions
withBeatNum:beatNumValue];

}
```

============== KickPattern - createKickPattern ==============

```
- (NSMutableArray*)createKickPattern:(NSMutableArray*)aEvenHitPos {

evenHitPos = aEvenHitPos;

kickHitPositions = [[NSMutableArray alloc]init];
kickPos0 = [[NSNumber alloc]initWithInt:0];
[kickHitPositions addObject:kickPos0];

for (n = 1; n <= ([evenHitPos count]-1) ; n++ ){
     if ((n % 2) != 0){
          [kickHitPositions addObject:[evenHitPos objectAtIndex:n]];
     }
}
return kickHitPositions;
}
```

======================== Hit - init ========================

```
- (id)init {


[super init];
```

```
order = 1;
position = 0;
center = 0;
eccentricity = 0;
intervalDuration = 0;

return self;
}
```

========== Hit - getOrder ==========

```
- (int)getOrder{
return order;
}
```

========== Hit - setOrder ==========

```
- (void)setOrder:(int)newOrder{
order = newOrder;
}
```

========== Hit - getHitPosition ==========

```
- (int)getHitPosition{
return position;
}
```

========== Hit - setHitPosition ==========

```
- (void)setHitPosition:(int)newHitPosition{
position = newHitPosition;
}
```

========== Hit - getCenter ==========

```
- (int)getCenter{
return center;
}
```

========== Hit - setCenter ==========

```
- (void)setCenter:(int)newCenter{
center = newCenter;
}
```

========== Hit - getEccentricity ==========

```
- (int)getEccentricity{


return eccentricity;
}
```

========== Hit - setEccentricity ==========

```
- (void)setEccentricity:(int)newEccentricity{
eccentricity = newEccentricity;
}
```

========== Hit - getIntervalDuration ==========

```
- (int)getIntervalDuration{
return intervalDuration;
}
```

========== Hit - setIntervalDuration ==========

```
- (void)setIntervalDuration:(int)newIntervalDuration{
intervalDuration = newIntervalDuration;
}
```

========== GUIController - designEvenSnarePattern ==========

```
- (void)designEvenSnarePattern{

// creo lo snarepattern
if (hitNumValue != 1){
     evenSnarePositions = [snarePattern createSnarePattern:evenHitPositions];}
else {
     evenSnarePositions = evenHitPositions;}

// assegnazione dell'ultimo colpo quando il numero totale di colpi è dispari
if (randKickSnare == 2){
     [evenSnarePositions addObject:[evenHitPositions objectAtIndex:hitNumValue]];}

// restore dello stato iniziale degli switch (tutti OFF)
for (n = 1; n <= beatNumValue ; n++){
     [[snareButtons objectAtIndex:n] setState:NSOffState];}
for (n = 1; n <= ((beatNumValue*3)/2) ; n++){
     [[snareTripletButtons objectAtIndex:n] setState:NSOffState];}

// disegna gli snareSwitches
for (n = 1; n <= ([evenSnarePositions count]-1) ; n++){
     int position = [[evenSnarePositions objectAtIndex:n] intValue];
     [[snareButtons objectAtIndex:position] setState:NSOnState];}

// misuro la variabilità del even Snare pattern
maxSnareShift = [patternCreator getVariability:evenSnarePositions
withBeatNum:beatNumValue];

}
```

========== GUIController - createSnarePattern ==========

```
- (NSMutableArray*)createSnarePattern:(NSMutableArray*)aEvenHitPos{

evenHitPos = aEvenHitPos;

snareHitPositions = [[NSMutableArray alloc]init];
snarePos0 = [[NSNumber alloc]initWithInt:0];
[snareHitPositions addObject:snarePos0];

for (n = 1; n <= ([evenHitPos count]-1) ; n++ ){
     if ((n % 2) == 0){
          [snareHitPositions addObject:[evenHitPos objectAtIndex:n]];}}
return snareHitPositions;
}

@end
```

========== GUIController - designHihatPattern ==========

```
- (void)designHihatPattern{

if (beatNumValue == 16){
     switch ([hh4816 intValue]){
          case 1:
               hhHitNumValue = 4;break;
          case 2:
               hhHitNumValue = 8;break;
          case 3:
               hhHitNumValue = 16;break;}}
if ((beatNumValue < 16) && (beatNumValue >= 8)){
     switch ([hh48 intValue]){
          case 1:
               hhHitNumValue = 4;break;
          case 2:
               hhHitNumValue = 8;break;}}
if ((beatNumValue < 8) && (beatNumValue >= 4)){
     hhHitNumValue = 4;}

// creo la distribuzione massima
hhEvenHitPositions = [patternCreator createEvenPattern:beatNumValue
withHits:hhHitNumValue];

//restore
for (n = 1; n <= beatNumValue; n++){
     [[hihatButtons objectAtIndex:n] setState:NSOffState];}
for (n = 1; n <= ((beatNumValue*3)/2); n++){
     [[hihatTripletButtons objectAtIndex:n] setState:NSOffState];}

// disegna gli hihatSwitches
for (n = 1; n <= ([hhEvenHitPositions count]-1) ; n++){
     int position = [[hhEvenHitPositions objectAtIndex:n] intValue];
     [[hihatButtons objectAtIndex:position] setState:NSOnState];
     if   (((hhHitNumValue == 16) &&
          ([[snareButtons objectAtIndex:position] state] == NSOnState)) ||


          ((hhHitNumValue == 8) && (beatNumValue < 16) && (beatNumValue >= 8) &&
          ([[snareButtons objectAtIndex:position] state] == NSOnState)) ||
          ((hhHitNumValue == 4) && (beatNumValue < 8) &&
          ([[snareButtons objectAtIndex:position] state] == NSOnState))){
               [[hihatButtons objectAtIndex:position] setState:NSOffState];}}

}
```

============ GUIController - designLTomPattern ============

```
- (void)designLTomPattern{
//restore
for (n = 1; n <= beatNumValue; n++){
     [[lTomButtons objectAtIndex:n] setState:NSOffState];}
for (n = 1; n <= ((beatNumValue*3)/2); n++){
     [[lTomTripletButtons objectAtIndex:n] setState:NSOffState];}
}
```

============ GUIController - designMTomPattern ============

```
- (void)designMTomPattern{
//restore
for (n = 1; n <= beatNumValue; n++){
     [[mTomButtons objectAtIndex:n] setState:NSOffState];}
for (n = 1; n <= ((beatNumValue*3)/2); n++){
     [[mTomTripletButtons objectAtIndex:n] setState:NSOffState];}
}
```

============ GUIController - designHTomPattern ============

```
- (void)designHTomPattern{
//restore
for (n = 1; n <= beatNumValue; n++){
     [[hTomButtons objectAtIndex:n] setState:NSOffState];}
for (n = 1; n <= ((beatNumValue*3)/2); n++){
     [[hTomTripletButtons objectAtIndex:n] setState:NSOffState];}
}
```

A.3 Le variazioni sul ritmo base

================ GUIController - varietyChange================

```
- (IBAction)varietyChange:(id)sender{

if ([hitNum intValue] != 1){
     [createRtmVariation setHidden:NO];
     [createEvenRtm setHidden:YES];}
}
```

============ GUIController - createRhythmVariation ============

```
- (IBAction)createRhythmVariation:(id)sender{

// impostazione della GUI
if (![stopButton isEnabled]){
     [playButton setEnabled:YES]; [bpm setEnabled:YES];}
if (![variety isEnabled]){
     [variety setEnabled:YES];}
[kickLock setEnabled:YES];  [snareLock setEnabled:YES];  [hihatLock setEnabled:YES];
[lTomLock setEnabled:YES];  [mTomLock setEnabled:YES];  [hTomLock setEnabled:YES];
[fillLength setEnabled:YES];  [generateFill setEnabled:YES];

// valori degli slider
beatNumValue = [beatNum intValue];
hitNumValue = [hitNum intValue];
varietyValue = [variety intValue];

// ritmo variato
rhythmVaried = YES;
```

```
// creazione delle variazioni in base allo stato dei Lock
if (([kickLock state] == NSOffState) && (hitNumValue > 2) && ([evenKickPositions
count]!=2)){
     [self designKickPatternVariation];}
if ([snareLock state] == NSOffState) {
     [self designSnarePatternVariation];}
if ([hihatLock state] == NSOffState) {
     [self designHihatPattern];}

}
```

========= GUIController - designKickPatternVariation =========

```
- (void)designKickPatternVariation{

// vario la distribuzione dei kick
if (varietyValue != 0){
     variedKickPositions = [patternCreator addVariation:varietyValue
toPositions:evenKickPositions];}
else {
     variedKickPositions = evenKickPositions;}

// restore dello stato iniziale degli switch (tutti OFF)
for (n = 1; n <= beatNumValue ; n++){
     [[kickButtons objectAtIndex:n] setState:NSOffState];}
for (n = 1; n <= ((beatNumValue*3)/2) ; n++){
     [[kickTripletButtons objectAtIndex:n] setState:NSOffState];}

// disegna i kickSwitches
for (n = 1; n <= ([variedKickPositions count]-1) ; n++){
     int position = [[variedKickPositions objectAtIndex:n] intValue];
     [[kickButtons objectAtIndex:position] setState:NSOnState];}

}
```

================ PatternCreator - addVariation ==============

```
- (NSMutableArray*)addVariation:(int)aVarietyValue
toPositions:(NSMutableArray*)aPatternPos{

     varietyValue = aVarietyValue;
     patternPos = aPatternPos;
     patternPosVar = [[NSMutableArray alloc] initWithArray:aPatternPos];

     int DistrCount;
     for ( DistrCount = 1; DistrCount <= varietyValue; DistrCount++) {
          int randHit = ((rand() % ([patternPosVar count]-1)) + 1);  // hitMaxNum
perch√® VOGLIO spostare anche il colpo numero 1
          int moved = 0;
          while ((moved == 0)) {

               while ([[patternPos objectAtIndex:randHit] intValue] == 1){
                    randHit = ((rand() % ([patternPosVar count]-1)) + 1);}

               pos = [[patternPosVar objectAtIndex:(randHit)] intValue];
               center = [[patternPos objectAtIndex:randHit] intValue];
               ecc = fabs(pos-center);
               prevPos = [[patternPosVar objectAtIndex:(randHit-1)] intValue];
               if ((randHit+1) > ([patternPosVar count]-1)){
                    nextPos = beatNumValue+1;}
               else {
                    nextPos = [[patternPosVar objectAtIndex:(randHit+1)] intValue];}

               NSLog (@"Sposto colpo da pos:%i ",pos);
```

```
            moved = 1;
            if ((((center - ecc) - prevPos)> 1) && ((nextPos - (center + ecc)) > 1)){
                int randval2 = rand() % 2;
                switch (randval2) {
                    case 0:
                        ecc++;
                        pos = (center - ecc);  break;
                    case 1:
                        ecc++;
                        pos = (center + ecc);  break;}}
            else if (((center - ecc) - prevPos) > 1){
                ecc++;
                pos = (center - ecc);}
            else if ((nextPos - (center + ecc)) > 1){
                ecc++;
                pos = (center + ecc);}
            else{
                moved = 0;
                randHit = ((rand() % ([patternPosVar count]-1)) + 1);}
        }       // fine while

        NSLog (@"a pos:%i ",pos);

        newPos = [[NSNumber alloc]initWithInt:pos];
        [patternPosVar replaceObjectAtIndex:randHit withObject:newPos];

    }// fine for

// Crea Array da passare al GUIController
    positions = [[NSMutableArray alloc]init];
    pos0 = [[NSNumber alloc]initWithInt:0];
    [positions addObject:pos0];
    int numHit = 1;
    for (n = 1; n <= beatNumValue ; n++){
        if (numHit <= ([patternPosVar count]-1)){
            if ((([[patternPosVar objectAtIndex:numHit] intValue]) == n){
                [positions addObject:[[NSNumber alloc]initWithInt:n]];
                numHit++;}}}

    return positions;
}

@end
```

========= GUIController - designSnarePatternVariation =========

```
- (void)designSnarePatternVariation{

// vario la distribuzione degli snare
if (varietyValue != 0){
    snareHitVarPositions = [patternCreator addVariation:varietyValue
toPositions:evenSnarePositions];}
else {
    snareHitVarPositions = evenSnarePositions;}

// restore dello stato iniziale degli switch (tutti OFF)
for (n = 1; n <= beatNumValue ; n++){
    [[snareButtons objectAtIndex:n] setState:NSOffState];}
for (n = 1; n <= ((beatNumValue*3)/2) ; n++){
    [[snareTripletButtons objectAtIndex:n] setState:NSOffState];}

// disegna gli snareSwitches
for (n = 1; n <= ([snareHitVarPositions count]-1) ; n++){
    int position = [[snareHitVarPositions objectAtIndex:n] intValue];
    [[snareButtons objectAtIndex:position] setState:NSOnState];}

}
```

A.4 Il Player

================== GUIController - play ==================

```
- (IBAction)play:(id)sender{

bpmValue = [bpm floatValue];
float timeInt = ((60 / bpmValue)/ 12); // 1/3 di 16esimo - 1/2 di terzina
timePos = 0;

kickSound95 = [[NSSound alloc] initWithContentsOfFile:@"../Samples/kick95.wav"
byReference:YES];
kickSound100 = [[NSSound alloc] initWithContentsOfFile:@"../Samples/kick100.wav"
byReference:YES];
kickSound105 = [[NSSound alloc] initWithContentsOfFile:@"../Samples/kick105.wav"
byReference:YES];
kickSound110 = [[NSSound alloc] initWithContentsOfFile:@"../Samples/kick110.wav"
byReference:YES];

snareSound95 = [[NSSound alloc] initWithContentsOfFile:@"../Samples/snare95.wav"
byReference:YES];
snareSound100 = [[NSSound alloc] initWithContentsOfFile:@"../Samples/snare100.wav"
byReference:YES];
snareSound105 = [[NSSound alloc] initWithContentsOfFile:@"../Samples/snare105.wav"
byReference:YES];
snareSound110 = [[NSSound alloc] initWithContentsOfFile:@"../Samples/snare110.wav"
byReference:YES];

hihatSound = [[NSSound alloc] initWithContentsOfFile:@"../Samples/hihat.wav"

byReference:YES];
hihatSound3 = [[NSSound alloc]
initWithContentsOfFile:@"../Samples/hihat8close100_1.wav" byReference:YES];

lTomSound80 = [[NSSound alloc] initWithContentsOfFile:@"../Samples/ltom80.wav"
byReference:YES];
lTomSound90 = [[NSSound alloc] initWithContentsOfFile:@"../Samples/ltom90.wav"
byReference:YES];
lTomSound105 = [[NSSound alloc] initWithContentsOfFile:@"../Samples/ltom105.wav"
byReference:YES];

mTomSound70 = [[NSSound alloc] initWithContentsOfFile:@"../Samples/mtom70.wav"
byReference:YES];
mTomSound82 = [[NSSound alloc] initWithContentsOfFile:@"../Samples/mtom82.wav"
byReference:YES];
mTomSound90 = [[NSSound alloc] initWithContentsOfFile:@"../Samples/mtom90.wav"
byReference:YES];

hTomSound85 = [[NSSound alloc] initWithContentsOfFile:@"../Samples/htom85.wav"
byReference:YES];
hTomSound105 = [[NSSound alloc] initWithContentsOfFile:@"../Samples/htom105.wav"
byReference:YES];
hTomSound120 = [[NSSound alloc] initWithContentsOfFile:@"../Samples/htom120.wav"
byReference:YES];


rhythmTimer = [NSTimer scheduledTimerWithTimeInterval:(timeInt) target:self
selector:@selector(playSound) userInfo:nil repeats:YES];

[stopButton setEnabled:YES];[playButton setEnabled:NO];
}
```

================== GUIController - playSound ==================

```
- (void)playSound{

// Loop
timePos = timePos+1;  timePosBin = (timePos+2)/3;  timePosTer = (timePos+1)/2;
if (timePos == ((beatNumValue*3)+1)){
     timePos = 1; timePosBin = 1; timePosTer = 1;
     [[kickButtons objectAtIndex:beatNumValue] setEnabled:YES];
     [[snareButtons objectAtIndex:beatNumValue] setEnabled:YES];
     [[hihatButtons objectAtIndex:beatNumValue] setEnabled:YES];
     [[lTomButtons objectAtIndex:beatNumValue] setEnabled:YES];
     [[mTomButtons objectAtIndex:beatNumValue] setEnabled:YES];
     [[hTomButtons objectAtIndex:beatNumValue] setEnabled:YES];
     [[kickTripletButtons objectAtIndex:((beatNumValue*3)/2)] setEnabled:YES];
     [[snareTripletButtons objectAtIndex:((beatNumValue*3)/2)] setEnabled:YES];
     [[hihatTripletButtons objectAtIndex:((beatNumValue*3)/2)] setEnabled:YES];
     [[lTomTripletButtons objectAtIndex:((beatNumValue*3)/2)] setEnabled:YES];
     [[mTomTripletButtons objectAtIndex:((beatNumValue*3)/2)] setEnabled:YES];
     [[hTomTripletButtons objectAtIndex:((beatNumValue*3)/2)] setEnabled:YES];
}
if (((timePos+2)%3) == 0){ //timePos di un sedicesimo
     [[kickButtons objectAtIndex:timePosBin] setEnabled:NO];
     [[snareButtons objectAtIndex:timePosBin] setEnabled:NO];
     [[hihatButtons objectAtIndex:timePosBin] setEnabled:NO];
     [[lTomButtons objectAtIndex:timePosBin] setEnabled:NO];
     [[mTomButtons objectAtIndex:timePosBin] setEnabled:NO];
     [[hTomButtons objectAtIndex:timePosBin] setEnabled:NO];
     [[kickButtons objectAtIndex:timePosBin-1] setEnabled:YES];
     [[snareButtons objectAtIndex:timePosBin-1] setEnabled:YES];
     [[hihatButtons objectAtIndex:timePosBin-1] setEnabled:YES];
     [[lTomButtons objectAtIndex:timePosBin-1] setEnabled:YES];
     [[mTomButtons objectAtIndex:timePosBin-1] setEnabled:YES];
     [[hTomButtons objectAtIndex:timePosBin-1] setEnabled:YES];

     kickHit = [[kickButtons objectAtIndex:timePosBin] state];
     snareHit = [[snareButtons objectAtIndex:timePosBin] state];
     hhHit = [[hihatButtons objectAtIndex:timePosBin] state];
     lTHit = [[lTomButtons objectAtIndex:timePosBin] state];
     mTHit = [[mTomButtons objectAtIndex:timePosBin] state];
     hTHit = [[hTomButtons objectAtIndex:timePosBin] state];
}
if (((timePos+1)%2) == 0){ //timePos di una terzina
     [[kickTripletButtons objectAtIndex:timePosTer] setEnabled:NO];
     [[snareTripletButtons objectAtIndex:timePosTer] setEnabled:NO];
     [[hihatTripletButtons objectAtIndex:timePosTer] setEnabled:NO];
     [[lTomTripletButtons objectAtIndex:timePosTer] setEnabled:NO];
     [[mTomTripletButtons objectAtIndex:timePosTer] setEnabled:NO];
     [[hTomTripletButtons objectAtIndex:timePosTer] setEnabled:NO];


     [[kickTripletButtons objectAtIndex:timePosTer-1] setEnabled:YES];
     [[snareTripletButtons objectAtIndex:timePosTer-1] setEnabled:YES];
     [[hihatTripletButtons objectAtIndex:timePosTer-1] setEnabled:YES];
     [[lTomTripletButtons objectAtIndex:timePosTer-1] setEnabled:YES];
     [[mTomTripletButtons objectAtIndex:timePosTer-1] setEnabled:YES];
     [[hTomTripletButtons objectAtIndex:timePosTer-1] setEnabled:YES];

     kickHit3 = [[kickTripletButtons objectAtIndex:timePosTer] state];
     snareHit3 = [[snareTripletButtons objectAtIndex:timePosTer] state];
     hhHit3 = [[hihatTripletButtons objectAtIndex:timePosTer] state];
     lTHit3 = [[lTomTripletButtons objectAtIndex:timePosTer] state];
     mTHit3 = [[mTomTripletButtons objectAtIndex:timePosTer] state];
     hTHit3 = [[hTomTripletButtons objectAtIndex:timePosTer] state];
}

if   (((kickHit == NSOnState) || (kickHit3 == NSOnState)) &&
     ([kickMute state] == NSOffState)){
```

```
     RandKickSound = (rand() % 4)+1;
     switch (RandKickSound){
          case 1:
               [kickSound95 stop];[kickSound95 play];
               kickHit = NSOffState;kickHit3 = NSOffState;break;//NSLog (@"95");
          case 2:
               [kickSound100 stop];[kickSound100 play];
               kickHit = NSOffState;kickHit3 = NSOffState;break;//NSLog (@"100");
          case 3:
               [kickSound105 stop];[kickSound105 play];
               kickHit = NSOffState;kickHit3 = NSOffState;break;//NSLog (@"105");
          case 4:
               [kickSound110 stop];[kickSound110 play];
               kickHit = NSOffState;kickHit3 = NSOffState;break;//NSLog (@"110");
     }//fine switch
}//fine if del kick

if   (((snareHit == NSOnState) || (snareHit3 == NSOnState)) &&
     ([snareMute state] == NSOffState)){
     RandSnareSound = (rand() % 4)+1;
     switch (RandSnareSound){
          case 1:
               [snareSound95 stop];[snareSound95 play];
               snareHit = NSOffState;snareHit3 = NSOffState;break;//NSLog (@"SN95");
          case 2:
               [snareSound100 stop];[snareSound100 play];
               snareHit = NSOffState;snareHit3 = NSOffState;break;//NSLog (@"SN100");
          case 3:
               [snareSound105 stop];[snareSound105 play];
               snareHit = NSOffState;snareHit3 = NSOffState;break;//NSLog (@"SN105");
          case 4:
               [snareSound110 stop];[snareSound110 play];
               snareHit = NSOffState;snareHit3 = NSOffState;break;//NSLog (@"SN110");
     }//fine switch
}//fine if di snare

if   (((hhHit == NSOnState) || (hhHit3 == NSOnState)) &&
     ([hihatMute state] == NSOffState)){
     [hihatSound stop];[hihatSound play];


     hhHit = NSOffState;hhHit3 = NSOffState;
}//fine if di hihat

if   (((lTHit == NSOnState) || (lTHit3 == NSOnState)) &&
     ([lTomMute state] == NSOffState)){
     RandLTomSound = (rand() % 3)+1;
     switch (RandLTomSound){
          case 1:
               [lTomSound80 stop];[lTomSound80 play];
               lTHit = NSOffState;lTHit3 = NSOffState;break;//NSLog (@"LT80");
          case 2:
               [lTomSound90 stop];[lTomSound90 play];
               lTHit = NSOffState;lTHit3 = NSOffState;break;//NSLog (@"LT90");
          case 3:
               [lTomSound105 stop];[lTomSound105 play];
               lTHit = NSOffState;lTHit3 = NSOffState;break;//NSLog (@"LT105");
     }//fine switch
}//fine if di low tom

if   (((mTHit == NSOnState) || (mTHit3 == NSOnState)) &&
     ([mTomMute state] == NSOffState)){
     RandMTomSound = (rand() % 3)+1;
     switch (RandMTomSound){
          case 1:
               [mTomSound70 stop];[mTomSound70 play];
               mTHit = NSOffState;mTHit3 = NSOffState;break;//NSLog (@"MT70");
          case 2:
```

```
                [mTomSound82 stop];[mTomSound82 play];
                mTHit = NSOffState;mTHit3 = NSOffState;break;//NSLog (@"MT82");
            case 3:
                [mTomSound90 stop];[mTomSound90 play];
                mTHit = NSOffState;mTHit3 = NSOffState;break;//NSLog (@"MT90");
        }//fine switch
}//fine if di mid tom

if   (((hTHit == NSOnState) || (hTHit3 == NSOnState)) &&
     ([hTomMute state] == NSOffState)){
     RandHTomSound = (rand() % 3)+1;
     switch (RandHTomSound){
          case 1:
               [hTomSound85 stop];[hTomSound85 play];
               hTHit = NSOffState;hTHit3 = NSOffState;break;//NSLog (@"HT85");
          case 2:
               [hTomSound105 stop];[hTomSound105 play];
               hTHit = NSOffState;hTHit3 = NSOffState;break;//NSLog (@"HT105");
          case 3:
               [hTomSound120 stop];[hTomSound120 play];
               hTHit = NSOffState;hTHit3 = NSOffState;break;//NSLog (@"HT120");
     }//fine switch
}//fine if di hit tom
}
```

================== GUIController - stop ==================

```
- (IBAction)stop:(id)sender{


[rhythmTimer invalidate];
[[kickButtons objectAtIndex:timePosBin] setEnabled:YES];
[[snareButtons objectAtIndex:timePosBin] setEnabled:YES];
[[hihatButtons objectAtIndex:timePosBin] setEnabled:YES];
[[lTomButtons objectAtIndex:timePosBin] setEnabled:YES];
[[mTomButtons objectAtIndex:timePosBin] setEnabled:YES];
[[hTomButtons objectAtIndex:timePosBin] setEnabled:YES];

[[kickTripletButtons objectAtIndex:timePosTer] setEnabled:YES];
[[snareTripletButtons objectAtIndex:timePosTer] setEnabled:YES];
[[hihatTripletButtons objectAtIndex:timePosTer] setEnabled:YES];
[[lTomTripletButtons objectAtIndex:timePosTer] setEnabled:YES];
[[mTomTripletButtons objectAtIndex:timePosTer] setEnabled:YES];
[[hTomTripletButtons objectAtIndex:timePosTer] setEnabled:YES];

[stopButton setEnabled:NO]; [playButton setEnabled:YES];

// release di tutti i campioni
[kickSound95 release];[kickSound100 release];
[kickSound105 release];[kickSound110 release];
[snareSound95 release];[snareSound100 release];
[snareSound105 release];[snareSound110 release];
[hihatSound release];[hihatSound3 release];
[lTomSound80 release];[lTomSound90 release];[lTomSound105 release];
[mTomSound70 release];[mTomSound82 release];[mTomSound90 release];
[hTomSound85 release];[hTomSound105 release];[hTomSound120 release];
}
```

A.5 Il generatore di fill

================ GUIController - createFill ================

```
- (IBAction)createFill:(id)sender{

if ([fillLength intValue] == 1){
     fillPos = [fillCreator composeFill:2];
     [self designFill:fillPos];}
else if ([fillLength intValue] == 2){
     fillPos = [fillCreator composeFill:4];
     [self designFill:fillPos];}
else if ([fillLength intValue] == 3){
     fillPos = [fillCreator composeFill:8];
     [self designFill:fillPos];}
}
```

================ FillCreator - composeFill ================

```
- (NSMutableArray*)composeFill:(int)aFillLength{

totalChunks = aFillLength;
fillPos = [[NSMutableArray alloc]init];
pos0 = [[NSNumber alloc]initWithInt:0];
[fillPos addObject:pos0];

chunk = [[NSArray alloc]init];

v = [[NSNumber alloc]initWithInt:0];
k = [[NSNumber alloc]initWithInt:1];
s = [[NSNumber alloc]initWithInt:2];
hh = [[NSNumber alloc]initWithInt:3];
t = [[NSNumber alloc]initWithInt:4];
t2 = [[NSNumber alloc]initWithInt:5];
t3 = [[NSNumber alloc]initWithInt:6];
b = [[NSNumber alloc]initWithInt:7];
```

```
for (n = 1; n <= totalChunks ; n++){
     chunk = [self createFillChunk:n withFillLength:totalChunks];
     [fillPos addObjectsFromArray:chunk];
}

return fillPos;
}
```

================ FillCreator - createFillChunk ================

```
- (NSArray*)createFillChunk:(int)aChunkNum withFillLength:(int)aFillLength2 {

chunk2 = [[NSMutableArray alloc]init];


chunkNum = aChunkNum;
fillLength2 = aFillLength2;

// settaggio percentuali per fill 1/4 chunk 1
if ((chunkNum == 1) && (fillLength2 == 1)){
     percXX = 73; percXO = 0; percOX = 0; percXXX = 27; percOO = 0;

     percSS = 55; percSK = 9; percST = 18;
     percKK = 0; percKS = 0; percKHh = 0; percKT = 9;
     percTT = 9;  percTTd = 0; percTK = 0; percTS = 0;
     percHhHh = 0; percHhS = 0; percHhK = 0; percHhT = 0;

     percS = 0;     percK = 0; percT = 0; percHh = 0;

     percOK = 0; percOS = 0;

     percSSS = 75; percSSK = 0; percSST = 0; percSKK = 0;
     percTTT = 25; percTTTd = 0; percTTuTd = 0; percTTK = 0; percTTKd = 0; percTKK = 0;
     percKKK = 0; percKKT = 0; percKKS = 0; percKSS = 0; percKTT = 0; percKTTd = 0;}

// settaggio percentuali per fill 1/4 chunk 2
if ((chunkNum == 2) && (fillLength2 == 1)){
     percXX = 47; percXO = 20; percOX = 0; percXXX = 33; percOO = 0;
     if ([[fillPos objectAtIndex:1]intValue] != 7){ // se il primo chunk non è binario
          percXX = 0; percXO = 47; percOX = 0; percXXX = 53; percOO = 0;}

     percSS = 50; percSK = 13; percST = 0;
     percKK = 13; percKS = 12; percKHh = 0; percKT = 0;
     percTT = 0;  percTTd = 12; percTK = 0; percTS = 0;
     percHhHh = 0; percHhS = 0; percHhK = 0; percHhT = 0;

     percS = 0;     percK = 67; percT = 33; percHh = 0;

     percOK = 0; percOS = 0;

     percSSS = 40; percSSK = 20; percSST = 20; percSKK = 0;
     percTTT = 0; percTTTd = 20; percTTuTd = 0; percTTK = 0; percTTKd = 0; percTKK = 0;
     percKKK = 0; percKKT = 0; percKKS = 0; percKSS = 0; percKTT = 0; percKTTd = 0;}

// settaggio percentuali per fill 2/4 chunk 1
if ((chunkNum == 1) && (fillLength2 == 2)){
     percXX = 53; percXO = 9; percOX = 0; percXXX = 38; percOO = 0;

     percSS = 88; percSK = 0; percST = 0;
     percKK = 0; percKS = 6; percKHh = 0; percKT = 0;
     percTT = 6;  percTTd = 0; percTK = 0; percTS = 0;
     percHhHh = 0; percHhS = 0; percHhK = 0; percHhT = 0;

     percS = 33;    percK = 0; percT = 67; percHh = 0;

     percOK = 0; percOS = 0;
```

```
     percSSS = 67; percSSK = 0; percSST = 0; percSKK = 0;
     percTTT = 33; percTTTd = 0; percTTuTd = 0; percTTK = 0; percTTKd = 0; percTKK = 0;
     percKKK = 0; percKKT = 0; percKKS = 0; percKSS = 0; percKTT = 0; percKTTd = 0;}


// settaggio percentuali per fill 2/4 chunk 2
if ((chunkNum == 2) && (fillLength2 == 2)){
     percXX = 69; percXO = 6; percOX = 0; percXXX = 25; percOO = 0;

     percSS = 55; percSK = 9; percST = 4;
     percKK = 9; percKS = 5; percKHh = 0; percKT = 0;
     percTT = 9;  percTTd = 0; percTK = 0; percTS = 9;
     percHhHh = 0; percHhS = 0; percHhK = 0; percHhT = 0;

     percS = 50;    percK = 0; percT = 50; percHh = 0;

     percOK = 0; percOS = 0;

     percSSS = 25; percSSK = 0; percSST = 0; percSKK = 25;
     percTTT = 25; percTTTd = 0; percTTuTd = 0; percTTK = 0; percTTKd = 0; percTKK = 0;
     percKKK = 0; percKKT = 0; percKKS = 0; percKSS = 0; percKTT = 0; percKTTd = 25;}

// settaggio percentuali per fill 2/4 chunk 3
if ((chunkNum == 3) && (fillLength2 == 2)){
     percXX = 81; percXO = 4; percOX = 0; percXXX = 15; percOO = 0;

     percSS = 64; percSK = 0; percST = 0;
     percKK = 0; percKS = 0; percKHh = 0; percKT = 0;
     percTT = 18;  percTTd = 0; percTK = 0; percTS = 0;
     percHhHh = 0; percHhS = 9; percHhK = 5; percHhT = 4;

     percS = 0;     percK = 0; percT = 100; percHh = 0;

     percOK = 0; percOS = 0;

     percSSS = 25; percSSK = 0; percSST = 0; percSKK = 0;
     percTTT = 75; percTTTd = 0; percTTuTd = 0; percTTK = 0; percTTKd = 0; percTKK = 0;
     percKKK = 0; percKKT = 0; percKKS = 0; percKSS = 0; percKTT = 0; percKTTd = 0;}

// settaggio percentuali per fill 2/4 chunk 4
if ((chunkNum == 4) && (fillLength2 == 2)){
     percXX = 32; percXO = 35; percOX = 0; percXXX = 20; percOO = 13;

     percSS = 40; percSK = 0; percST = 0;
     percKK = 0; percKS = 0; percKHh = 0; percKT = 0;
     percTT = 20;  percTTd = 20; percTK = 20; percTS = 0;
     percHhHh = 0; percHhS = 0; percHhK = 0; percHhT = 0;

     percS = 19;    percK = 27; percT = 45; percHh = 9;

     percOK = 0; percOS = 0;

     percSSS = 17; percSSK = 17; percSST = 0; percSKK = 0;
     percTTT = 33; percTTTd = 0; percTTuTd = 0; percTTK = 17; percTTKd = 0; percTKK =
16;
     percKKK = 0; percKKT = 0; percKKS = 0; percKSS = 0; percKTT = 0; percKTTd = 0;}

// settaggio percentuali per fill 4/4 chunk 1
if ((chunkNum == 1) && (fillLength2 == 3)){
     percXX = 56; percXO = 15; percOX = 0; percXXX = 29; percOO = 0;



     percSS = 63; percSK = 0; percST = 0;
     percKK = 16; percKS = 11; percKHh = 0; percKT = 0;
     percTT = 10;  percTTd = 0; percTK = 0; percTS = 0;
```

```
     percHhHh = 0; percHhS = 0; percHhK = 0; percHhT = 0;

     percS = 60; percK = 40; percT = 0; percHh = 0;

     percOK = 0; percOS = 0; percOHh = 0; percOT = 0;

     percSSS = 30; percSSK = 20; percSST = 0; percSKK = 0;
     percTTT = 30; percTTTd = 0; percTTuTd = 0; percTTK = 0; percTTKd = 0; percTKK = 0;
     percKKK = 0; percKKT = 0; percKKS = 10; percKSS = 10; percKTT = 0; percKTTd = 0;}

// settaggio percentuali per fill 4/4 chunk 2
if ((chunkNum == 2) && (fillLength2 == 3)){
     percXX = 40; percXO = 12; percOX = 11; percXXX = 37; percOO = 0;

     percSS = 51; percSK = 7; percST = 7;
     percKK = 0; percKS = 14; percKHh = 7; percKT = 0;
     percTT = 0;  percTTd = 7; percTK = 0; percTS = 0;
     percHhHh = 0; percHhS = 7; percHhK = 0; percHhT = 0;

     percS = 50;    percK = 50; percT = 0; percHh = 0;

     percOK = 50; percOS = 50; percOHh = 0; percOT = 0;

     percSSS = 23; percSSK = 0; percSST = 0; percSKK = 31;
     percTTT = 15; percTTTd = 0; percTTuTd = 0; percTTK = 8; percTTKd = 0; percTKK = 8;
     percKKK = 0; percKKT = 0; percKKS = 0; percKSS = 7; percKTT = 8; percKTTd = 0;}

// settaggio percentuali per fill 4/4 chunk 3
if ((chunkNum == 3) && (fillLength2 == 3)){
     percXX = 53; percXO = 12; percOX = 6; percXXX = 29; percOO = 0;

     percSS = 78; percSK = 6; percST = 0;
     percKK = 0; percKS = 6; percKHh = 0; percKT = 0;
     percTT = 5;  percTTd = 5; percTK = 0; percTS = 0;
     percHhHh = 0; percHhS = 0; percHhK = 0; percHhT = 0;

     percS = 100;   percK = 0; percT = 0; percHh = 0;

     percOK = 0; percOS = 50; percOHh = 0; percOT = 50;

     percSSS = 10; percSSK = 0; percSST = 0; percSKK = 0;
     percTTT = 50; percTTTd = 0; percTTuTd = 0; percTTK = 20; percTTKd = 0; percTKK =
0;
     percKKK = 0; percKKT = 10; percKKS = 0; percKSS = 0; percKTT = 10; percKTTd = 0;}

// settaggio percentuali per fill 4/4 chunk 4
if ((chunkNum == 4) && (fillLength2 == 3)){
     percXX = 44; percXO = 15; percOX = 0; percXXX = 41; percOO = 0;

     percSS = 53; percSK = 0; percST = 0;
     percKK = 13; percKS = 13; percKHh = 0; percKT = 0;
     percTT = 21;  percTTd = 0; percTK = 0; percTS = 0;
     percHhHh = 0; percHhS = 0; percHhK = 0; percHhT = 0;


     percS = 60;    percK = 40; percT = 0; percHh = 0;

     percOK = 0; percOS = 0; percOHh = 0; percOT = 0;

     percSSS = 22; percSSK = 7; percSST = 0; percSKK = 7;
     percTTT = 14; percTTTd = 0; percTTuTd = 0; percTTK = 0; percTTKd = 7; percTKK =
29;
     percKKK = 0; percKKT = 0; percKKS = 0; percKSS = 0; percKTT = 14; percKTTd = 0;}

// settaggio percentuali per fill 4/4 chunk 5
if ((chunkNum == 5) && (fillLength2 == 3)){
     percXX = 43; percXO = 9; percOX = 6; percXXX = 42; percOO = 0;
```

```
     percSS = 47; percSK = 7; percST = 7;
     percKK = 7; percKS = 20; percKHh = 6; percKT = 0;
     percTT = 0;  percTTd = 0; percTK = 0; percTS = 0;
     percHhHh = 0; percHhS = 6; percHhK = 0; percHhT = 0;

     percS = 100;  percK = 0; percT = 0; percHh = 0;

     percOK = 100; percOS = 0; percOHh = 0; percOT = 0;

     percSSS = 47; percSSK = 0; percSST = 0; percSKK = 0;
     percTTT = 33; percTTTd = 0; percTTuTd = 0; percTTK = 7; percTTKd = 0; percTKK = 0;
     percKKK = 0; percKKT = 7; percKKS = 0; percKSS = 6; percKTT = 0; percKTTd = 0;}

// settaggio percentuali per fill 4/4 chunk 6
if ((chunkNum == 6) && (fillLength2 == 3)){
     percXX = 46; percXO = 12; percOX = 6; percXXX = 36; percOO = 0;

     percSS = 80; percSK = 0; percST = 0;
     percKK = 0; percKS = 7; percKHh = 7; percKT = 0;
     percTT = 0;  percTTd = 0; percTK = 0; percTS = 6;
     percHhHh = 0; percHhS = 0; percHhK = 0; percHhT = 0;

     percS = 100; percK = 0; percT = 0; percHh = 0;

     percOK = 0; percOS = 0; percOHh = 50; percOT = 50;

     percSSS = 34; percSSK = 0; percSST = 0; percSKK = 8;
     percTTT = 17; percTTTd = 0; percTTuTd = 0; percTTK = 0; percTTKd = 0; percTKK =
33;
     percKKK = 0; percKKT = 0; percKKS = 0; percKSS = 0; percKTT = 8; percKTTd = 0;}

// settaggio percentuali per fill 4/4 chunk 7
if ((chunkNum == 7) && (fillLength2 == 3)){
     percXX = 59; percXO = 6; percOX = 0; percXXX = 32; percOO = 3;

     percSS = 80; percSK = 0; percST = 0;
     percKK = 0; percKS = 5; percKHh = 0; percKT = 0;
     percTT = 15;  percTTd = 0; percTK = 0; percTS = 0;
     percHhHh = 0; percHhS = 0; percHhK = 0; percHhT = 0;

     percS = 100;  percK = 0; percT = 0; percHh = 0;

     percOK = 0; percOS = 0; percOHh = 0; percOT = 0;


     percSSS = 18; percSSK = 0; percSST = 9; percSKK = 0;
     percTTT = 28; percTTTd = 0; percTTuTd = 9; percTTK = 0; percTTKd = 0; percTKK = 0;
     percKKK = 27; percKKT = 9; percKKS = 0; percKSS = 0; percKTT = 0; percKTTd = 0;}

// settaggio percentuali per fill 4/4 chunk 8
if ((chunkNum == 8) && (fillLength2 == 3)){
     percXX = 29; percXO = 32; percOX = 0; percXXX = 21; percOO = 18;

     percSS = 70; percSK = 0; percST = 0;
     percKK = 0; percKS = 0; percKHh = 0; percKT = 10;
     percTT = 0;  percTTd = 0; percTK = 10; percTS = 10;
     percHhHh = 0; percHhS = 0; percHhK = 0; percHhT = 0;

     percS = 45;   percK = 19; percT = 36; percHh = 0;

     percOK = 0; percOS = 0; percOHh = 0; percOT = 0;

     percSSS = 29; percSSK = 0; percSST = 0; percSKK = 0;
     percTTT = 43; percTTTd = 0; percTTuTd = 14; percTTK = 0; percTTKd = 0; percTKK =
14;
     percKKK = 0; percKKT = 0; percKKS = 0; percKSS = 0; percKTT = 0; percKTTd = 0;}
// calcolo del range dei pattern di chunk
minXX = 0;        maxXX = percXX;
```

```
minXO = maxXX;    maxXO = maxXX+percXO;
minOX = maxXO;    maxOX = maxXO+percOX;
minXXX = maxOX;   maxXXX = maxOX+percXXX;
minOO = maxXXX;   maxOO = maxXXX+percOO;
// calcolo del range degli instruments del chunk XX
minSS = 0;    maxSS = percSS;
minSK = maxSS;    maxSK = maxSS+percSK;
minST = maxSK; maxST = maxSK+percST;
minKK = maxST;    maxKK = maxST+percKK;
minKS = maxKK;    maxKS = maxKK+percKS;
minKHh = maxKS;   maxKHh = maxKS+percKHh;
minKT = maxKHh;   maxKT = maxKHh+percKT;
minTT = maxKT;    maxTT = maxKT+percTT;
minTTd = maxTT;   maxTTd = maxTT+percTTd;
minTK = maxTTd;   maxTK = maxTTd+percTK;
minTS = maxTK;    maxTS = maxTK+percTS;
minHhHh = maxTS; maxHhHh = maxTS+percHhHh;
minHhS = maxHhHh; maxHhS = maxHhHh+percHhS;
minHhK = maxHhS; maxHhK = maxHhS+percHhK;
minHhT = maxHhK; maxHhT = maxHhK+percHhT;
// calcolo del range degli instruments del chunk XO
minS = 0;maxS = percS;     minK = maxS;maxK = maxS+percK;
minT = maxK;maxT = maxK+percT; minHh = maxT;maxHh = maxT+percHh;
// calcolo del range degli instruments del chunk OX
minOS = 0;maxOS = percOS;
minOK = maxOS;maxOK = maxOS+percOK;
minOHh = maxOK;maxOHh = maxOK+percOHh;
minOT = maxOHh;maxOT = maxOHh+percOT;
// calcolo del range degli instruments del chunk XXX
minSSS = 0;maxSSS = percSSS;
minSSK = maxSSS;maxSSK = maxSSS+percSSK;
minSST = maxSSK;maxSST = maxSSK+percSST;
minSKK = maxSST;maxSKK = maxSST+percSKK;


minTTT = maxSKK;maxTTT = maxSKK+percTTT;
minTTTd = maxTTT;maxTTTd = maxTTT+percTTTd;
minTTuTd = maxTTTd;maxTTuTd = maxTTTd+percTTuTd;
minTTK = maxTTuTd;maxTTK = maxTTuTd+percTTK;
minTTKd = maxTTK;maxTTKd = maxTTK+percTTKd;
minTKK = maxTTKd;maxTKK = maxTTKd+percTKK;
minKKK = maxTKK;maxKKK = maxTKK+percKKK;
minKKT = maxKKK;maxKKT = maxKKK+percKKT;
minKKS = maxKKT;maxKKS = maxKKT+percKKS;
minKSS = maxKKS;maxKSS = maxKKS+percKSS;
minKTT = maxKSS;maxKTT = maxKSS+percKTT;
minKTTd = maxKTT;maxKTTd = maxKTT+percKTTd;

// calcolo del valore casuale di chunk pattern e instrument
randChunkPatt = (rand() % 100)+1;   // valori percentuali da 0 a 100
randChunkInst = (rand() % 100)+1;   // valori percentuali da 0 a 100

if ((percXX != 0) && (randChunkPatt > minXX) && (randChunkPatt <= maxXX)){ // colpi XX
     if ((randChunkInst > minSS) && (randChunkInst <= maxSS)){
          [chunk2 addObject:b];[chunk2 addObject:s];[chunk2 addObject:s];}
     if ((randChunkInst > minSK) && (randChunkInst <= maxSK)){
          [chunk2 addObject:b];[chunk2 addObject:s];[chunk2 addObject:k];}
     if ((randChunkInst > minST) && (randChunkInst <= maxST)){
          [chunk2 addObject:b];[chunk2 addObject:s];[chunk2 addObject:t];}
     if ((randChunkInst > minKK) && (randChunkInst <= maxKK)){
          [chunk2 addObject:b];[chunk2 addObject:k];[chunk2 addObject:k];}
     if ((randChunkInst > minKS) && (randChunkInst <= maxKS)){
          [chunk2 addObject:b];[chunk2 addObject:k];[chunk2 addObject:s];}
     if ((randChunkInst > minKHh) && (randChunkInst <= maxKHh)){
          [chunk2 addObject:b];[chunk2 addObject:k];[chunk2 addObject:hh];}
     if ((randChunkInst > minKT) && (randChunkInst <= maxKT)){
          [chunk2 addObject:b];[chunk2 addObject:k];[chunk2 addObject:t];}
     if ((randChunkInst > minTT) && (randChunkInst <= maxTT)){
```

```
          [chunk2 addObject:b];[chunk2 addObject:t];[chunk2 addObject:t];}
     if ((randChunkInst > minTTd) && (randChunkInst <= maxTTd)){
          [chunk2 addObject:b];[chunk2 addObject:t];[chunk2 addObject:t2];}
     if ((randChunkInst > minTK) && (randChunkInst <= maxTK)){
          [chunk2 addObject:b];[chunk2 addObject:t];[chunk2 addObject:k];}
     if ((randChunkInst > minTS) && (randChunkInst <= maxTS)){
          [chunk2 addObject:b];[chunk2 addObject:t];[chunk2 addObject:s];}
     if ((randChunkInst > minHhHh) && (randChunkInst <= maxHhHh)){
          [chunk2 addObject:b];[chunk2 addObject:hh];[chunk2 addObject:hh];}
     if ((randChunkInst > minHhS) && (randChunkInst <= maxHhS)){
          [chunk2 addObject:b];[chunk2 addObject:hh];[chunk2 addObject:s];}
     if ((randChunkInst > minHhK) && (randChunkInst <= maxHhK)){
          [chunk2 addObject:b];[chunk2 addObject:hh];[chunk2 addObject:k];}
     if ((randChunkInst > minHhT) && (randChunkInst <= maxHhT)){
          [chunk2 addObject:b];[chunk2 addObject:hh];[chunk2 addObject:t];}
}

if ((percXO != 0) && (randChunkPatt > minXO) && (randChunkPatt <= maxXO)){ // colpi XO
     if ((randChunkInst > minS) && (randChunkInst <= maxS)){
          [chunk2 addObject:b];[chunk2 addObject:s];[chunk2 addObject:v];}
     if ((randChunkInst > minK) && (randChunkInst <= maxK)){
          [chunk2 addObject:b];[chunk2 addObject:k];[chunk2 addObject:v];}
     if ((randChunkInst > minT) && (randChunkInst <= maxT)){


          [chunk2 addObject:b];[chunk2 addObject:t];[chunk2 addObject:v];}
     if ((randChunkInst > minHh) && (randChunkInst <= maxHh)){
          [chunk2 addObject:b];[chunk2 addObject:hh];[chunk2 addObject:v];}
}

if ((percOX != 0) && (randChunkPatt > minOX) && (randChunkPatt <= maxOX)){ // colpi OX
     if ((randChunkInst > minOS) && (randChunkInst <= maxOS)){
          [chunk2 addObject:b];[chunk2 addObject:v];[chunk2 addObject:s];}
     if ((randChunkInst > minOK) && (randChunkInst <= maxOK)){
          [chunk2 addObject:b];[chunk2 addObject:v];[chunk2 addObject:k];}
     if ((randChunkInst > minOHh) && (randChunkInst <= maxOHh)){
          [chunk2 addObject:b];[chunk2 addObject:v];[chunk2 addObject:hh];}
     if ((randChunkInst > minOT) && (randChunkInst <= maxOT)){
          [chunk2 addObject:b];[chunk2 addObject:v];[chunk2 addObject:t];}
}

if ((percXXX != 0) && (randChunkPatt > minXXX) && (randChunkPatt <= maxXXX)){ // colpi
XXX
     if ((randChunkInst > minSSS) && (randChunkInst <= maxSSS)){
          [chunk2 addObject:s];[chunk2 addObject:s];[chunk2 addObject:s];}
     if ((randChunkInst > minSSK) && (randChunkInst <= maxSSK)){
          [chunk2 addObject:s];[chunk2 addObject:s];[chunk2 addObject:k];}
     if ((randChunkInst > minSST) && (randChunkInst <= maxSST)){
          [chunk2 addObject:s];[chunk2 addObject:s];[chunk2 addObject:t];}
     if ((randChunkInst > minSKK) && (randChunkInst <= maxSKK)){
          [chunk2 addObject:s];[chunk2 addObject:k];[chunk2 addObject:k];}
     if ((randChunkInst > minTTT) && (randChunkInst <= maxTTT)){
          [chunk2 addObject:t];[chunk2 addObject:t];[chunk2 addObject:t];}
     if ((randChunkInst > minTTTd) && (randChunkInst <= maxTTTd)){
          [chunk2 addObject:t];[chunk2 addObject:t2];[chunk2 addObject:t3];}
     if ((randChunkInst > minTTuTd) && (randChunkInst <= maxTTuTd)){
          [chunk2 addObject:t];[chunk2 addObject:t];[chunk2 addObject:t2];}
     if ((randChunkInst > minTTK) && (randChunkInst <= maxTTK)){
          [chunk2 addObject:t];[chunk2 addObject:t];[chunk2 addObject:k];}
     if ((randChunkInst > minTTKd) && (randChunkInst <= maxTTKd)){
          [chunk2 addObject:t];[chunk2 addObject:t2];[chunk2 addObject:k];}
     if ((randChunkInst > minTKK) && (randChunkInst <= maxTKK)){
          [chunk2 addObject:t];[chunk2 addObject:k];[chunk2 addObject:k];}
     if ((randChunkInst > minKKK) && (randChunkInst <= maxKKK)){
          [chunk2 addObject:k];[chunk2 addObject:k];[chunk2 addObject:k];}
     if ((randChunkInst > minKKT) && (randChunkInst <= maxKKT)){
          [chunk2 addObject:k];[chunk2 addObject:k];[chunk2 addObject:t];}
     if ((randChunkInst > minKKS) && (randChunkInst <= maxKKS)){
```

```
          [chunk2 addObject:k];[chunk2 addObject:k];[chunk2 addObject:s];}
     if ((randChunkInst > minKSS) && (randChunkInst <= maxKSS)){
          [chunk2 addObject:k];[chunk2 addObject:s];[chunk2 addObject:s];}
     if ((randChunkInst > minKTT) && (randChunkInst <= maxKTT)){
          [chunk2 addObject:k];[chunk2 addObject:t];[chunk2 addObject:t];}
     if ((randChunkInst > minKTTd) && (randChunkInst <= maxKTTd)){
          [chunk2 addObject:k];[chunk2 addObject:t];[chunk2 addObject:t2];}
}
return chunk2;
}

@end
```

================ GUIController - designFill ================

```
- (void)designFill:(NSMutableArray*)afillPos{

int position,val1, val2, val3;

if ([fillLength intValue] == 1){

// Cancellazione colpi fill 1/4
     int m = 13;
     for (n = 19; n <= ((beatNumValue*3)/2); n++){
          if (m <= beatNumValue){
               [[kickButtons objectAtIndex:m] setState:NSOffState];
               [[snareButtons objectAtIndex:m] setState:NSOffState];
               [[hihatButtons objectAtIndex:m] setState:NSOffState];
               if ([lTomLock state] == NSOffState){
                    [[lTomButtons objectAtIndex:m] setState:NSOffState];}
               if ([mTomLock state] == NSOffState){
                    [[mTomButtons objectAtIndex:m] setState:NSOffState];}
               if ([hTomLock state] == NSOffState){
                    [[hTomButtons objectAtIndex:m] setState:NSOffState];}
               m++;
          }
          [[kickTripletButtons objectAtIndex:n] setState:NSOffState];
          [[snareTripletButtons objectAtIndex:n] setState:NSOffState];
          [[hihatTripletButtons objectAtIndex:n] setState:NSOffState];
          if ([lTomLock state] == NSOffState){
               [[lTomTripletButtons objectAtIndex:n] setState:NSOffState];}
          if ([mTomLock state] == NSOffState){
               [[mTomTripletButtons objectAtIndex:n] setState:NSOffState];}
          if ([hTomLock state] == NSOffState){
               [[hTomTripletButtons objectAtIndex:n] setState:NSOffState];}
     }
// Disegno del fill
     for (n = 1; n <= 2; n++){
          if ([[fillPos objectAtIndex:((n*3)-2)]intValue] == 7){ // binario
               position = 12+(((n-1)*2)+1);
               val1 = [[fillPos objectAtIndex:((n*3)-1)]intValue];
               val2 = [[fillPos objectAtIndex:(n*3)]intValue];
               [self designBinChunk:position firstValue:val1 secondValue:val2];}
          if ([[fillPos objectAtIndex:((n*3)-2)]intValue] != 7){ // ternario
               position = 18+(((n-1)*3)+1);
               val1 = [[fillPos objectAtIndex:((n*3)-2)]intValue];
               val2 = [[fillPos objectAtIndex:((n*3)-1)]intValue];
               val3 = [[fillPos objectAtIndex:(n*3)]intValue];
               [self designTerChunk:position firstValue:val1 secondValue:val2
thirdValue:val3];}
     }
}// fine if - fill corto

else if ([fillLength intValue] == 2){
```

```
// Cancellazione colpi fill 2/4
     int m = 9;
     for (n = 13; n <= ((beatNumValue*3)/2); n++){


     if (m <= beatNumValue){
               [[kickButtons objectAtIndex:m] setState:NSOffState];
               [[snareButtons objectAtIndex:m] setState:NSOffState];
               [[hihatButtons objectAtIndex:m] setState:NSOffState];
               if ([lTomLock state] == NSOffState){
                    [[lTomButtons objectAtIndex:m] setState:NSOffState];}
               if ([mTomLock state] == NSOffState){
                    [[mTomButtons objectAtIndex:m] setState:NSOffState];}
               if ([hTomLock state] == NSOffState){
                    [[hTomButtons objectAtIndex:m] setState:NSOffState];}
               m++;
          }
          [[kickTripletButtons objectAtIndex:n] setState:NSOffState];
          [[snareTripletButtons objectAtIndex:n] setState:NSOffState];
          [[hihatTripletButtons objectAtIndex:n] setState:NSOffState];
          if ([lTomLock state] == NSOffState){
               [[lTomTripletButtons objectAtIndex:n] setState:NSOffState];}
          if ([mTomLock state] == NSOffState){
               [[mTomTripletButtons objectAtIndex:n] setState:NSOffState];}
          if ([hTomLock state] == NSOffState){
               [[hTomTripletButtons objectAtIndex:n] setState:NSOffState];}
          }
// Disegno del fill
     for (n = 1; n <= 4; n++){
          if ([[fillPos objectAtIndex:((n*3)-2)]intValue] == 7){ // binario
               position = 8+(((n-1)*2)+1);
               val1 = [[fillPos objectAtIndex:((n*3)-1)]intValue];
               val2 = [[fillPos objectAtIndex:(n*3)]intValue];
               [self designBinChunk:position firstValue:val1 secondValue:val2];}
          if ([[fillPos objectAtIndex:((n*3)-2)]intValue] != 7){ // ternario
               position = 12+(((n-1)*3)+1);
               val1 = [[fillPos objectAtIndex:((n*3)-2)]intValue];
               val2 = [[fillPos objectAtIndex:((n*3)-1)]intValue];
               val3 = [[fillPos objectAtIndex:(n*3)]intValue];
               [self designTerChunk:position firstValue:val1 secondValue:val2
thirdValue:val3];}
     }
}// fine if - fill medio

else if ([fillLength intValue] == 3){

// Cancellazione colpi fill 4/4
     int m = 1;
     for (n = 1; n <= ((beatNumValue*3)/2); n++){
          if (m <= beatNumValue){
               [[kickButtons objectAtIndex:m] setState:NSOffState];
               [[snareButtons objectAtIndex:m] setState:NSOffState];
               [[hihatButtons objectAtIndex:m] setState:NSOffState];
               if ([lTomLock state] == NSOffState){
                    [[lTomButtons objectAtIndex:m] setState:NSOffState];}
               if ([mTomLock state] == NSOffState){
                    [[mTomButtons objectAtIndex:m] setState:NSOffState];}
               if ([hTomLock state] == NSOffState){
                    [[hTomButtons objectAtIndex:m] setState:NSOffState];}
               m++;
          }


          [[kickTripletButtons objectAtIndex:n] setState:NSOffState];
          [[snareTripletButtons objectAtIndex:n] setState:NSOffState];
          [[hihatTripletButtons objectAtIndex:n] setState:NSOffState];
          if ([lTomLock state] == NSOffState){
               [[lTomTripletButtons objectAtIndex:n] setState:NSOffState];}
```

```
          if ([mTomLock state] == NSOffState){
               [[mTomTripletButtons objectAtIndex:n] setState:NSOffState];}
          if ([hTomLock state] == NSOffState){
               [[hTomTripletButtons objectAtIndex:n] setState:NSOffState];}
     }
// Disegno del fill
     for (n = 1; n <= 8; n++){
          if ([[fillPos objectAtIndex:((n*3)-2)]intValue] == 7){ // binario
               position = (((n-1)*2)+1);
               val1 = [[fillPos objectAtIndex:((n*3)-1)]intValue];
               val2 = [[fillPos objectAtIndex:(n*3)]intValue];
               [self designBinChunk:position firstValue:val1 secondValue:val2];}
          if ([[fillPos objectAtIndex:((n*3)-2)]intValue] != 7){ // ternario
               position = (((n-1)*3)+1);
               val1 = [[fillPos objectAtIndex:((n*3)-2)]intValue];
               val2 = [[fillPos objectAtIndex:((n*3)-1)]intValue];
               val3 = [[fillPos objectAtIndex:(n*3)]intValue];
               [self designTerChunk:position firstValue:val1 secondValue:val2
thirdValue:val3];}
     }
}// fine if - fill lungo
}
```

============== GUIController - designBinChunk ==============

```
- (void)designBinChunk:(int)aPos firstValue:(int)aVal1 secondValue:(int)aVal2{

     int pos = aPos;
     int val1 = aVal1;
     int val2 = aVal2;

     switch (val1){
          case 0:  break;
          case 1:  [[kickButtons objectAtIndex:pos] setState:NSOnState];break;
          case 2:  [[snareButtons objectAtIndex:pos] setState:NSOnState];break;
          case 3: [[hihatButtons objectAtIndex:pos] setState:NSOnState];break;
          case 4: [[lTomButtons objectAtIndex:pos] setState:NSOnState];break;
          case 5: [[mTomButtons objectAtIndex:pos] setState:NSOnState];break;
          case 6: [[hTomButtons objectAtIndex:pos] setState:NSOnState];break;}
     switch (val2){
          case 0: break;
          case 1: [[kickButtons objectAtIndex:(pos+1)] setState:NSOnState];break;
          case 2: [[snareButtons objectAtIndex:(pos+1)] setState:NSOnState];break;
          case 3: [[hihatButtons objectAtIndex:(pos+1)] setState:NSOnState];break;
          case 4: [[lTomButtons objectAtIndex:(pos+1)] setState:NSOnState];break;
          case 5: [[mTomButtons objectAtIndex:(pos+1)] setState:NSOnState];break;
          case 6: [[hTomButtons objectAtIndex:(pos+1)] setState:NSOnState];break;}
}
```

============== GUIController - designTerChunk ==============

```
- (void)designTerChunk:(int)aPos firstValue:(int)aVal1 secondValue:(int)aVal2
thirdValue:(int)aVal3{

int pos = aPos;
int val1 = aVal1;
int val2 = aVal2;
int val3 = aVal3;

switch (val1){
     case 0:  break;
     case 1:  [[kickTripletButtons objectAtIndex:pos] setState:NSOnState];break;
     case 2:  [[snareTripletButtons objectAtIndex:pos] setState:NSOnState];break;
```

```
    case 3: [[hihatTripletButtons objectAtIndex:pos] setState:NSOnState];break;
    case 4: [[lTomTripletButtons objectAtIndex:pos] setState:NSOnState];break;
    case 5: [[mTomTripletButtons objectAtIndex:pos] setState:NSOnState];break;
    case 6: [[hTomTripletButtons objectAtIndex:pos] setState:NSOnState];break;}
switch (val2){
    case 0: break;
    case 1: [[kickTripletButtons objectAtIndex:(pos+1)] setState:NSOnState];break;
    case 2: [[snareTripletButtons objectAtIndex:(pos+1)] setState:NSOnState];break;
    case 3: [[hihatTripletButtons objectAtIndex:(pos+1)] setState:NSOnState];break;
    case 4: [[lTomTripletButtons objectAtIndex:(pos+1)] setState:NSOnState];break;
    case 5: [[mTomTripletButtons objectAtIndex:(pos+1)] setState:NSOnState];break;
    case 6: [[hTomTripletButtons objectAtIndex:(pos+1)] setState:NSOnState];break;}
switch (val3){
    case 0: break;
    case 1: [[kickTripletButtons objectAtIndex:(pos+2)] setState:NSOnState];break;
    case 2: [[snareTripletButtons objectAtIndex:(pos+2)] setState:NSOnState];break;
    case 3: [[hihatTripletButtons objectAtIndex:(pos+2)] setState:NSOnState];break;
    case 4: [[lTomTripletButtons objectAtIndex:(pos+2)] setState:NSOnState];break;
    case 5: [[mTomTripletButtons objectAtIndex:(pos+2)] setState:NSOnState];break;
    case 6: [[hTomTripletButtons objectAtIndex:(pos+2)] setState:NSOnState];break;}
}

@end
```

BIBLIOGRAFIA

1. G. D'Anna. *Dizionario italiano ragionato*. Sintesi. 1989. p. 1557.

2. *The Compact Edition of the Oxford English Dictionary*. **II**. Oxford University Press. 1971. p. 2537.

3. *Enciclopedia della musica.* Garzanti. 2005. p. 753.

4. Simha Arom. *African Polyphony and Polyrhythm*. Cambridge University Press, Cambridge, England, 1991.

5. Laz E. N. Ekwueme. *Concepts in African musical theory*. Journal of Black Studies, 5(1):35-64, September 1974.

6. Douglas Eck. *A positive-evidence model for classifying rhythmical patterns*. Technical Report IDSIA-09-00, Instituto Dalle Molle di studi sull'intelligenza artificiale, Manno, Switzerland, 2000.

7. O. Wright. *The Modal System of Arab and Persian Music AD 1250-1300*. Oxford University Press, Oxford, England, 1978.

8. Kjell Gustafson. *A new method for displaying speech rhythm, with illustrations from some Nordic languages*. In K. Gregersen and H. Basb ll, editors, Nordic Prosody IV, pages 105-114. Odense University Press, 1987.

9. Kjell Gustafson. *The graphical representation of rhythm*. In (PROPH) Progress Reports from Oxford Phonetics, volume 3, pages 6-26, University of Oxford, 1988.

10. Ludger Hofmann-Engl. *Rhythmic similarity: A theoretical and empirical approach*. In C. Stevens, D. Burnham, G. McPherson, E. Schubert, and J. Renwick, editors, Proceedings of the Seventh International Conference on Music Perception and Cognition, pages 564-567, Sidney, Australia, 2002.

11. Godfried T. Toussaint. *A mathematical analysis of African, Brazilian, and Cuban clave rhythms*. In Proceedings of BRIDGES: Mathematical Connections in Art, Music and Science, pages 157-168, Towson University, Towson, MD, July 27-29 2002.

12. Godfried T. Toussaint. *Classification and phylogenetic analysis of African ternary rhythm timelines*. In Proceedings of BRIDGES: Mathematical Connections in Art, Music and Science, pages 25-36, Granada, Spain, July 23-27 2003.

13. M. Chemillier. *Ethnomusicology, ethnomathematics. The logic underlying orally transmitted artistic practices*. In G. Assayag, H. G. Feichtinger, and J. F. Rodrigues, editors, Mathematics and Music, pages 161-183. Springer, 2002.

14. Godfried T. Toussaint. *A mathematical measure of preference in African rhythm.* In Abstracts of Papers Presented to the American Mathematical Society, volume 25, page 248, Phoenix, January 7-10 2004. American Mathematical Society.

15. A. Forte. *The Structure of Atonal Music*. Yale Univ. Press, New Haven, 1973.

16. Steven Block and Jack Douthett. *Vector products and intervallic weighting*. Journal of Music Theory, 38:21-41, 1994.

17. L. Fejes T oth. *On the sum of distances determined by a pointset*. Acta. Math. Acad. Sci. Hungar., 7:397-401, 1956.

18. Brian J. McCartin. *Prelude to musical geometry*. The College Mathematics Journal, 29(5):354-370, 1998.

19. David Locke. Drum Gahu: *An Introduction to African Rhythm*. White Cliffs Media, Gilsum, New Hampshire, 1998.

20. Michael Keith. *From Polychords to P olya: Adventures in Musical Combinatorics*. Vinculum Press, Princeton, 1991.

21. Frank Ruskey and Joe Sawada. *An efficient algorithm for generating necklaces with fixed density*. SIAM Journal of Computing, 29(2):671-684, 1999.

22. Paul Lemke, Steven S. Skiena, and Warren D. Smith. *Reconstructing sets from interpoint distances*. Tech. Rept. DIMACS-2002-37, 2002.

23. C. Chieh. *Analysis of cyclotomic sets. Zeitschrift Kristallographie*, 150:261-277, 1979.

24. Sung-Hyuk Cha and Sargur N. Srihari. *On measuring the distance between histograms*. Pattern Recognition, 35:1355-1370, 2002.

25. Rainer Typke, Panos Giannopoulos, Remco C. Veltkamp, Frans Wiering, and Ren e van Oostrum. *Using transportation distances for measuring melodic similarity*. In Holger H. Hoos and David Bainbridge, editors, Proceedings of the Fourth International Symposium on Music Information Retrieval, pages 107-114, Johns Hopkins University, Baltimore, 2003.

26. R. W. Hamming. *Coding and Information Theory*. Prentice-Hall, Englewood Cli s, 1986.

27. Bernard Mont-Reynaud and Malcolm Goldstein. *On finding rhythmic patterns in musical lines*. In Proceedings of the International Computer Music Conference, pages 391-397, San Francisco, California, 1985.

28. E. J. Coyle and I. Shmulevich. *A system for machine recognition of music patterns.* In Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing, Seattle, Washington, 1998.

29. Ilya Shmulevich, Olli Yli-Harja, Edward Coyle, Dirk-Jan Povel, and Kjell Lemstr om. *Perceptual issues in music pattern recognition*. Computers and the Humanities, 35:23-35, 2001.

30. Keith S. Orpen and David Huron. *Measurement of similarity in music: A quantitative approach for non-parametric representations.* In Computers in Music Research, volume 4, pages 1-44. 1992.

31. M. Mongeau and D. Sanko. *Comparison of musical sequences.* Computers and the Humanities, 24:161-175, 1990.

32. Abraham Bookstein, Shmuel Tomi Klein, and Timo Raita. *Fuzzy Hamming distance:*

a new dissimilarity measure. In Proceedings 12th Annual Symposium on Combinatorial Pattern Matching, volume LNCS 2089, pages 86-97, Berlin, 2001. Springer-Verlag.

33. Abraham Bookstein, Vladimir A. Kulyukin, and Timo Raita. *Generalized Hamming* distance. Information Retrieval, 5(4):353-375, 2002.

34. Greg Aloupis, Prosenjit Bose, Erik D. Demaine, Stefan Langerman, Henk Meijer, Mark Overmars, and Godfried T. Toussaint. *Computing signed permutations of polygons.* In Proc. 14th Canadian Conf. Computational Geometry, pages 68-71, Univ. of Lethbridge, Alberta, 2002.

35. Greg Aloupis, Erik D. Demaine, Henk Meijer, Joseph O'Rourke, Ileana Streinu, and Godfried T. Toussaint. *On flat-state connectivity of chains with fixed acute angles*. In Proc. 14th Canadian Conf. Computational Geometry, pages 27-30, Univ. of Lethbridge, Alberta, 2002.

36. N. G. de Bruijn. *Sorting by means of swapping*. Discrete Mathematics, 9:333-339, 1974.

37. Therese Biedl, Timothy Chan, Erik D. Demaine, Rudolf Fleischer, Mordecai Golin, James A. King, and Ian Munro. *Fun-sort - or the chaos of unordered binary search*.Discrete Applied Mathematics. to appear.

38. Therese Biedl, Timothy Chan, Erik D. Demaine, Rudolf Fleischer, Mordecai Golin, and J. Ian Munro. Fun-sort. In Elena Lodi, Linda Pagli, and Nicola Santoro, editors, *Proceedings of 2nd International Conference on Fun with Algorithms* (FUN 2001), pages 15-26, Isola d'Elba, Italy, May 29-31 2001.

39. Adam Berenzweig, Daniel P. W. Ellis, and Steve Lawrence. *Anchor space for classification and similarity measurement of music*. In Proceedings ICME-03, Baltimore, U.S.A., July 2003.

40. Godfried T. Toussaint. *Sharper lower bounds for discrimination information in terms of variation*. IEEE Transactions on Information Theory, pages 99-100, January 1975.

41. Greg Aloupis, Thomas Fevens, Stefan Langerman, Tomomi Matsui, Antonio Mesa, Yurai Nu~nez, David Rappaport, and Godfried Toussaint. *Computing a geometric measure of the similarity between two melodies*. In Proc. 15th Canadian Conf. Computational Geometry, pages 81-84, Dalhousie University, Halifax, Nova Scotia, Canada, August 11-13 2003.

42. J. Clough and J. Douthett. *Maximally even sets.* journal of Music Theory, 35:93-173, 1991.

43. Gregory J. Chaitin. *Information theoretic computational complexity.* IEEE Transactions on Information Theory, IT-20:10-15, 1974.

44. Abraham Moles. *Information Theory and Esthetics Perception*. The University of Illinois Press, Urbana and London, 1966.

45. A. Lempel and J. Ziv. *On the complexity of finite sequences.* IEEE Transactions on Information Theory, IT-22(1):75-81, 1976

46. Jeff Pressing. *Cognitive complexity and the structure of musical patterns*. In Proceedings of the 4th Conference of the Australasian Cognitive Science Society, Newcastle, Australia, 1997.

47. F. Lerdahl and R. Jackendoff. *A generative Theory of Tonal Music.* MIT Press, Cambridge, Massachussetts, 1983.

48. Ronald C. Read. *Combinatorial problems in the theory of music.* Discrete Mathematics, 167-168, April 1997.

49. Joel K. Haak. *Clapping Music - a combinatorial problem.* The College Mathematics Journal, 22:224-227, May 1991.

50. Joel K. Haak. *The Mathematics of Steve Reich's Clapping Music.* In proceedings of BRIDGES: Mathematical Connections in Art, Music and Science, pages 87-92, Winfield, Kansas, 1998.

51. Justin London. *Hierarchical representations of complex meters.* In 6th International Conference on Music, Perception and Cognition, Keele University, United Kingdom, August 5-10 2000.

52. Steve Reich. *Writings about Music.* The Press of the Nova Scotia College of Art and Design, Halifax, Canada, 1974.