Corsi Accademici di Musica Elettronica DCPL34
Conservatorio A. Casella, L'Aquila
GAMMA

Giulio Romano De Mattia
esame di **Composizione Musicale Informatica**

27/06/2025

Sommario

Il presente scritto documenta il lavoro svolto nell'arco di un anno e mezzo in merito al brano di composizione algoritmica *Gamma*. Non volendomi fermare alla contemplazione del risultato musicale in quanto tale, in questa tesina porrò l'attenzione sugli strumenti compositivi scritti per la realizzazione del brano poiché reputo lo strumento stesso e l'ambiente di sviluppo digitale creato come fondamenta della composizione, se non composizione essa stessa. Verrà così esplorato il motore di csound, ultimo elemento della catena, per risalire poi al codice python, per arrivare fin sù la sorgente, il dizionario YAML che rappresenta la partitura della composizione: il *sorgente* di Gamma.

Indice

1 INTRODUZIONE

Il presente lavoro documenta lo sviluppo di *Gamma*, un sistema compositivo algoritmico che rappresenta una tappa fondamentale nel più ampio progetto del ciclo *Delta*. Questa tesina nasce dall'esigenza di formalizzare e analizzare un percorso compositivo che, partito con ambizioni di complessità adattiva, ha rivelato la necessità di un passaggio intermedio attraverso un sistema di composizione algoritmica.

1.1 Il Ciclo Delta e la Genesi di Gamma

Il ciclo compositivo *Delta* nasce dalla volontà di studiare come modellare un sistema musicale complesso adattivo. Concepito come sistema chiuso e acusmatico, Delta rappresenta un passo preliminare e necessario prima di approcciare lo studio e la realizzazione di ecosistemi performativi aperti, come quelli esplorati da compositori come Agostino Di Scipio. La scelta di lavorare inizialmente con un sistema chiuso non è limitativa, ma strategica: permette di concentrarsi sulla comprensione e modellazione delle dinamiche interne senza le variabili aggiuntive dell'interazione in tempo reale con l'ambiente o con i performer.

La sfida principale che ha portato alla nascita di Gamma non risiedeva nel mantenere un controllo compositivo bensì nello studio in vitro ovvero in una dimensione estremamente controllate di uno degli agenti che stavo costruendo per il sistema complesso Delta. Ho ritenuto utile scrivere prima un brano preparatorio utilizzando esclusivamente uno dei sistemi messi in relazione in Delta, ovvero lo strumento 'Comportamento' che in Gamma prende il nome di 'Voce'. 'Voce' ha all'interno un comportamento caotico derivato dall'utilizzo di una mappa logistica in feedback che utilizzo per ricavarmi i ritmi futuri (vedremo in seguito nella tesina e discuteremo i vari casi).

Gamma è un laboratorio dove sperimentare, comprendere e affinare gli strumenti compositivi prima di lanciarsi nella complessità delle dinamiche caotiche e adattive.

La scelta del nome *Gamma* riflette precisamente questa funzione: rappresenta una gamma di possibilità esperibili da Delta. Se Delta è la foce dove tutti i flussi convergono attraverso reti di relazioni, Gamma è la sorgente - il luogo dove questi flussi nascono chiari e distinguibili, dove è possibile osservare e comprendere ogni singolo rivolo prima che si mescoli con gli altri.

1.2 Dalle Note alle Nuvole: Un'Eredità Compositiva

Gamma si inserisce nella tradizione della composizione sistematica ispirandosi liberamente ai metodi di lavoro sviluppati da Iannis Xenakis e Barry Truax. L'approccio delle *maschere di tendenza* che caratterizza il sistema non è un'interpretazione personale e un'implementazione specifica di tecniche già consolidate nella letteratura della computer music. Xenakis aveva esplorato l'uso di distribuzioni probabilistiche per la generazione di masse sonore. Truax sviluppò l'approccio delle maschere di tendenza per necessità pratiche legate alla sin-

tesi granulare: quando si lavora con tecniche che richiedono la generazione di milioni di parametri per controllare nuvole di grani sonori, diventa impossibile specificare ogni singolo valore. Le maschere di tendenza emergono quindi come soluzione naturale per gestire questa complessità, permettendo di definire comportamenti statistici globali piuttosto che valori individuali.

Ciò che Gamma apporta a questa tradizione è una sistematizzazione particolare di questi concetti. Il sistema implementa quattro modalità distinte di generazione parametrica (range, choices, distribuzione normale, valore fisso), organizzate in una gerarchia compositiva chiara (composizione → sezioni → layer → eventi). Questa strutturazione permette di gestire la complessità mantenendo un controllo compositivo semplice.

L'uso delle maschere di tendenza in Gamma permette di lavorare su diversi livelli di astrazione simultaneamente. A livello micro, si possono definire distribuzioni precise per singoli parametri; a livello macro, si possono creare evoluzioni graduali attraverso l'interpolazione tra stati.

1.3 Struttura e Obiettivi della Tesina

Il presente lavoro si propone di documentare e analizzare il sistema Gamma sotto molteplici prospettive, fornendo sia una comprensione teorica dei principi sottostanti sia una guida pratica all'implementazione e all'uso del sistema.

Gli obiettivi principali sono:

1. **Documentare l'architettura del sistema**: Fornire una descrizione dettagliata e sistematica di tutti i componenti software che costituiscono Gamma, dalle strutture dati Python agli strumenti Csound, dalla sintassi YAML al sistema di visualizzazione.
2. **Valutare criticamente il sistema**: far risaltare attraverso la spiegazoine i punti di forza e i punti deboli del alvoro, sia dal punto di vista tecnico che estetico, fornendo spunti per sviluppi futuri.
3. **Preparare il terreno per Delta**: Comprendere come l'esperienza di Gamma informi e prepari lo sviluppo del sistema adattivo completo previsto per Delta.

2 L'ORCHESTRA GAMMA

L'orchestra Csound di Gamma non è un sistema autonomo, ma il motore di sintesi e di esecuzione progettato specificamente per interpretare le strutture musicali complesse generate dallo script Python `generative_composer.py`. Ogni strumento e opcode è stato creato per tradurre in suono un parametro o un comportamento definito nel file YAML di input. Lo strumento Voce funge da *ponte* principale, ricevendo un intero *comportamento* (un cluster di

eventi) da Python e orchestrandone la micro-temporalità e la sintesi. In questo capitolo, analizzeremo come questa traduzione avviene, partendo dal livello macroscopico (Voce) fino al dettaglio del singolo evento (eventoSonoro).

2.1 Lo Strumento Voce: Generatore di Comportamenti

Lo strumento `Voce` costituisce il livello più alto della gerarchia di sintesi in Gamma. Non genera direttamente suoni, ma orchestra la creazione di sequenze di eventi sonori secondo logiche compositive complesse. La sua definizione inizia con una ricca parametrizzazione:

```
instr Voce
    ; ------------------------------------------------------------------------
    ; 1. INIZIALIZZAZIONE E ACQUISIZIONE PARAMETRI
    ; ------------------------------------------------------------------------
    i_CAttacco       = p2             ; Tempo di attacco del comportamento
    i_Durata         = p3             ; Durata complessiva
    i_RitmiTab       = p4             ; Tabella dei ritmi
    i_DurataArmonica = p5             ; Durata armonica di riferimento
    i_DynamicIndex   = p6
    i_Ottava         = p7
    i_Registro       = p8
    i_ottava_arrivo = p9
    i_registro_arrivo = p10
    i_PosTab         = p11             ; Tabella delle posizioni
    i_IdComp         = p12            ; ID del comportamento
    i_NonlinearMode  = (p13 == 0 ? 3 : p13)
    i_SensoMovimento = (p14 == 0 ? 1 : p14)
    i_ifnAttacco     = (p15 == 0 ? 10 : p15)
    i_ifn_section_env = p16
    i_section_start_time = p17
    i_duration_leeway = p19
    i_section_duration = p18 + p19
    i_section_end = i_section_start_time + i_section_duration
    iSafetyBuffer = p20
```

Ogni parametro ha un significato musicale preciso:

- `i_CAttacco` e `i_Durata`: definiscono la finestra temporale in cui il comportamento è attivo
- `i_RitmiTab`: punta a una tabella contenente la sequenza di valori ritmici che determinano sia la temporalità che le frequenze degli eventi
- `i_DurataArmonica`: il valore di riferimento per il calcolo delle durate reali degli eventi
- `i_Ottava` e `i_Registro`: coordinate nello spazio delle altezze di partenza
- `i_ottava_arrivo` e `i_registro_arrivo`: destinazione per eventuali glissandi
- `i_NonlinearMode`: seleziona l'algoritmo di generazione per nuovi ritmi

2.1.1 *Gestione Adattiva della Durata e dei Confini di Sezione*

Un aspetto cruciale per la coerenza musicale è la gestione degli eventi che superano i confini della loro sezione. Il parametro iSafetyBuffer attiva una logica di controllo fondamentale:

```
    if i_EventAttack + i_EventDuration > i_section_end then
        if iSafetyBuffer == 1 then
            i_EventDuration = i_section_end - i_EventAttack + random:i(0,
              ↪ i_duration_leeway)
        endif
    endif
```

Quando un evento sta per *sforare* la fine della sezione (definita da `i_section_end`), la sua durata viene troncata per terminare esattamente al confine. Inoltre, viene aggiunto un piccolo tempo casuale (`i_duration_leeway`) per evitare che tutti gli eventi terminino bruscamente allo stesso istante, creando una fine più organica e meno artificiale. Questa logica, controllata da Python, è essenziale per assemblare sezioni consecutive senza creare sovrapposizioni o troncamenti sonori indesiderati (qualora si manteng il safety buffer attivo).

2.1.2 *Il Loop Generativo Principale*

Il cuore dello strumento Voce è un loop while che genera eventi fino al raggiungimento della durata specificata:

```
i_EventIdx = 0
i_whileTime = 0

while i_whileTime < i_Durata do
    ; -------- 3.1 GESTIONE RITMI --------
    if i_EventIdx < i_LenRitmiTab then
        i_RitmoCorrente tab_i i_EventIdx, i_TempRitmiTab
        if i_RitmoCorrente == 0 then
            goto generateNewRhythm
        endif
        i_Vecchio_Ritmo = (i_EventIdx == 0) ? 1 : tab_i(i_EventIdx - 1,
          ↪ i_TempRitmiTab)
    else
        generateNewRhythm:
        i_Vecchio_Ritmo tab_i i_EventIdx - 1, i_TempRitmiTab
        i_RitmoCorrente NonlinearFunc i_Vecchio_Ritmo, i_NonlinearMode
        tabw_i i_RitmoCorrente, i_EventIdx, i_TempRitmiTab
    endif
```

Questo codice implementa una logica sofisticata: inizialmente legge i ritmi dalla tabella fornita, ma quando questa si esaurisce, genera nuovi valori usando l'opcode `NonlinearFunc`, creando potenzialmente sequenze infinite che evolvono secondo regole caotiche o deterministiche.

2.1.3 Calcolo Temporale degli Eventi

Il timing di ogni evento dipende dal ritmo precedente secondo la formula:

```
if i_EventIdx == 0 then
    i_EventAttack = i_CAttacco
else
    i_RitmoNormalizzato = 1 / i_Vecchio_Ritmo
    i_PreviousAttack tab_i gi_Index - 1, gi_eve_attacco
    i_EventAttack = i_DurataArmonica * i_RitmoNormalizzato + i_PreviousAttack
endif
```

Questa relazione inversamente proporzionale significa che valori ritmici più alti producono eventi più ravvicinati, creando accelerazioni, mentre valori bassi generano rarefazioni temporali.

2.1.4 Gestione della Tabella Ritmi Temporanea

Una caratteristica importante è la creazione di una tabella temporanea estesa per i ritmi:

```
i_LenRitmiTab = ftlen(i_RitmiTab)
i_TempRitmiTab ftgen 0, 0, i_LenRitmiTab + 10000, -2, 0

; Copia i ritmi iniziali nella tabella temporanea
i_IndexCopy = 0
while i_IndexCopy < i_LenRitmiTab do
    i_ValRitmo tab_i i_IndexCopy, i_RitmiTab
    tabw_i i_ValRitmo, i_IndexCopy, i_TempRitmiTab
    i_IndexCopy += 1
od
```

Questo approccio permette di estendere dinamicamente la sequenza ritmica oltre i valori iniziali senza modificare la tabella originale, mantenendo la purezza dei dati di input mentre si esplora lo spazio generativo.

2.1.5 Sistema di Scheduling degli Eventi

La creazione effettiva degli eventi sonori avviene attraverso la chiamata a `schedule`:

```
schedule "eventoSonoro", i_EventAttack - p2, i_EventDuration, i_DynamicIndex,
    ↪ i_Freq1, i_Pos, i_RitmoCorrente, i_Freq2, i_ifnAttacco, gi_Index, i_IdComp,
    ↪  i_SensoMovimento, i_ifn_section_env, i_section_start_time,
    ↪ i_section_duration
```

Il parametro `i_RitmoCorrente` viene passato come `p7` allo strumento eventoSonoro, dove viene letto come `iHR` che spiegherò in seguito.

2.2 EventoSonoro: Dal Parametro al Suono

Lo strumento `eventoSonoro` è responsabile della generazione effettiva del suono. Riceve i parametri calcolati da Voce e li trasforma in segnale audio attraverso sintesi e processamento.

2.2.1 Sistema di Compensazione Isofonica dell'Ampiezza

Una delle caratteristiche più sofisticate di Gamma è l'implementazione di un sistema di calibrazione dell'ampiezza basato sulle curve isofoniche dello standard ISO 226:2003[1]. Per comprendere l'importanza di questa implementazione, è necessario esaminare il fenomeno psicoacustico che la motiva.

L'orecchio umano non percepisce tutte le frequenze con la stessa sensibilità. Un tono puro a 100 Hz deve avere un'intensità fisica significativamente maggiore di un tono a 3000 Hz per essere percepito con la stessa loudness. Le curve isofoniche mappano questa non-linearità percettiva, mostrando quali livelli di pressione sonora (SPL) sono necessari a diverse frequenze per produrre la stessa sensazione di loudness.

Lo standard ISO 226:2003 rappresenta la revisione più recente di queste curve, basata su estesi studi psicoacustici internazionali. Ogni curva rappresenta un livello di loudness costante misurato in phon, dove per definizione:

- A 1000 Hz, il livello in phon equivale al livello in dB SPL
- A tutte le altre frequenze, il livello in dB SPL necessario varia secondo la curva

Il sistema utilizza tre tabelle fondamentali derivate dallo standard ISO:

```
1 giIsoFreqs ftgen 0, 0, 32, -2, 20, 25, 31.5, 40, 50, 63, 80, 100, 125, 160, 200,
    ↪  250, 315, 400, 500, 630, 800, 1000, 1250, 1600, 2000, 2500, 3150, 4000,
    ↪ 5000, 6300, 8000, 10000, 12500
2 giAf       ftgen 0, 0, 32, -2, 0.532, 0.506, 0.480, 0.455, 0.432, 0.409, 0.387,
    ↪ 0.367, 0.349, 0.330, 0.315, 0.301, 0.288, 0.276, 0.267, 0.259, 0.253,
    ↪ 0.250, 0.246, 0.244, 0.243, 0.243, 0.243, 0.242, 0.242, 0.245, 0.254,
    ↪ 0.271, 0.301
3 giLu       ftgen 0, 0, 32, -2, -31.6, -27.2, -23.0, -19.1, -15.9, -13.0, -10.3,
    ↪ -8.1, -6.2, -4.5, -3.1, -2.0, -1.1, -0.4, 0.0, 0.3, 0.5, 0.0, -2.7, -4.1,
    ↪ -1.0, 1.7, 2.5, 1.2, -2.1, -7.1, -11.2, -10.7, -3.1
4 giTf       ftgen 0, 0, 32, -2, 78.5, 68.7, 59.5, 51.1, 44.0, 37.5, 31.5, 26.5,
    ↪ 22.1, 17.9, 14.4, 11.4, 8.6, 6.2, 4.4, 3.0, 2.2, 2.4, 3.5, 1.7, -1.3, -4.2,
    ↪  -6.0, -5.4, -1.5, 6.0, 12.6, 13.9, 12.3
```

Questi parametri rappresentano:

- **giAf**: Esponente di loudness, determina la pendenza della funzione di trasferimento
- **giLu**: Livello di loudness alla soglia, rappresenta la correzione per la soglia uditiva

- **giTf**: Soglia uditiva in campo libero, il livello minimo udibile in condizioni ideali

Il calcolo dell'ampiezza compensata avviene in più fasi:

```
kamp GetIsoAmp_k i_DynamicIndex, ifreq1, ifreq2
```

Questo UDO k-rate gestisce la compensazione durante i glissandi. Per frequenze statiche, il calcolo è più diretto:

```
opcode GetIsoAmp, i, ii
    iFrequency, iDynamicIndex xin
    iSafeFrequency = limit(iFrequency, 20, 12500)

; 1. Recupera i parametri di base per la dinamica data
    iPhonLevel, iDbfsRef1kHz GetDynamicParams iDynamicIndex

; 2. Calcola il dB SPL target per la frequenza e il livello phon dati
    iDbSplTarget    PhonToSpl_i     iPhonLevel, iSafeFrequency

; 3. Il dB SPL di riferimento a 1kHz è per definizione uguale al livello Phon
    iDbSplRef1kHz   =               iPhonLevel

; 4. Calcola l'offset di compensazione
    iFrequencyOffset = iDbSplTarget - iDbSplRef1kHz

; 5. Applica l'offset al livello dBFS di riferimento
    iFinalDbfs      = iDbfsRef1kHz + iFrequencyOffset

; 6. Converti il dBFS finale in ampiezza lineare
    iFinalAmp       = ampdbfs(iFinalDbfs)

xout iFinalAmp
endop
```

Invece di applicare curve di equalizzazione complesse, il sistema calcola quanto la frequenza target si discosta dal riferimento a 1kHz e applica questa differenza al livello dBFS desiderato.

La conversione da phon a SPL implementa la formula matematica dello standard:

```
opcode PhonToSpl_i, i, ii
    iphon, ifreq    xin

; Interpolazione lineare dalle tabelle ISO
    iaf             Interp  ifreq, giIsoFreqs, giAf
    ilu             Interp  ifreq, giIsoFreqs, giLu
    itf             Interp  ifreq, giIsoFreqs, giTf

; Formula ISO 226:2003
    iterm1          =       4.47e-3 * (pow(10, 0.025 * iphon) - 1.15)
    iterm2_exp      =       (itf + ilu) / 10.0 - 9
    iterm2          =       pow(0.4 * pow(10, iterm2_exp), iaf)
    iaf_value       =       iterm1 + iterm2

if iaf_value <= 0 then
        ispl        =       itf + (iphon / 40.0) * 20
```

```
    else
        ispl        =       (10.0 / iaf) * log10(iaf_value) - ilu + 94.0
    endif

if abs(ifreq - 1000) < 0.1 then
        ispl = iphon
    endif

xout            ispl
endop
```

La formula si divide in due termini:

- **iterm1**: Rappresenta la componente lineare della loudness, dominante a livelli alti
- **iterm2**: Cattura la non-linearità vicino alla soglia uditiva

Il caso speciale `if iaf_value` $\leqslant$ `0` gestisce situazioni vicine o sotto la soglia uditiva, dove la formula principale potrebbe produrre valori matematicamente indefiniti.

Questa implementazione garantisce che:

1. **Coerenza Percettiva**: Un evento marcato come `mf` (mezzoforte) mantiene la stessa loudness percepita indipendentemente dalla sua frequenza
2. **Glissandi Naturali**: Durante un glissando, l'ampiezza viene continuamente aggiustata per compensare i cambiamenti di sensibilità dell'orecchio
3. **Bilanciamento Automatico**: In texture polifoniche, eventi in registri diversi mantengono bilanciamento percettivo senza intervento manuale

Per esempio, un evento a 100 Hz marcato come `f` (forte) riceverà automaticamente più energia di uno a 3000 Hz con la stessa dinamica, compensando la minore sensibilità dell'orecchio alle basse frequenze. Questa compensazione è particolarmente critica nel sistema pitagorico di Gamma, dove le frequenze generate possono spaziare su tutto lo spettro udibile.

L'implementazione k-rate per i glissandi assicura che questa compensazione avvenga continuamente:

```
if iAmpStart > iAmpEnd then
    kf expseg  1, p3, 0.0001
    kFinalAmp = (kf * (iAmpStart-iAmpEnd))+iAmpEnd
elseif iAmpStart < iAmpEnd then
    kf expseg  0.0001, p3, 1
    kFinalAmp = (kf * (iAmpEnd-iAmpStart))+iAmpStart
```

L'uso di segmenti esponenziali invece che lineari è per la natura logaritmica della percezione dell'ampiezza, creando transizioni che appaiono lineari all'ascolto.

2.2.2 Sistema di Spazializzazione Mid-Side e Armoniche Spaziali

La spazializzazione in Gamma va oltre il semplice panning stereofonico, implementando un sistema basato su *armoniche spaziali* di mia ideazione che deriva dalla teoria delle armoniche ritmiche. Il concetto chiave è che i valori ritmici non solo organizzano il tempo e selezionano le frequenze, ma definiscono anche il movimento nello spazio stereofonico.

Vediamo come si sviluppa questo sistema partendo dai parametri di base:

```
; Parametri di base per la spazializzazione
iwhichZero = abs(p6)    ; quale "zero" della funzione trigonometrica usare
iHR = max(1, abs(p7))   ; Harmonic Ratio - il numero di "spicchi" della
   ↪ circonferenza

; Calcolo del periodo e della posizione iniziale
iPeriod = $M_PI * 2 / iHR
iradi = (iwhichZero > 0 ? (iwhichZero - 1) * iPeriod : 0)
```

Il parametro `iHR` (Harmonic Ratio) determina in quanti *spicchi* viene suddivisa la circonferenza. Ad esempio:

- `iHR = 1`: un solo periodo, movimento completo 0-360°
- `iHR = 4`: quattro periodi, la circonferenza è divisa in quadranti
- `iHR = 7`: sette spicchi, creando una suddivisione asimmetrica

Il parametro `iwhichZero` determina da quale zero della funzione trigonometrica iniziare il movimento:

```
; Evoluzione temporale della posizione angolare
kndx_local line 0, p3, 1
ktab tab kndx_local, ifn_shape, 1
krad = iradi + (ktab * iPeriod * i_senso)
```

Qui `krad` evolve nel tempo secondo l'inviluppo specificato da `ifn_shape`, modulato dal senso di movimento (`i_senso` = 1 o -1 per movimento orario/antiorario).

La generazione dell'inviluppo locale usa una modifica della funzione seno quando `ifn_shape == 2`:

```
if ifn_shape == 2 then
    kEnv_local = abs(sin(krad * iHR / 2))
else
    kEnv_local tab kndx_local, ifn_shape, 1
endif
```

La formula `abs(sin(krad * iHR / 2))` genera curve polari modificate. Questa trasformazione:

- Prende il valore assoluto, creando lobi sempre positivi
- Moltiplica per `iHR / 2`, dimezzando il numero di lobi rispetto agli spicchi spaziali
- Crea una correlazione diretta tra movimento spaziale e ampiezza

Per comprendere meglio, consideriamo il codice Python fornito che visualizza queste funzioni:

```
def genera_e_plotta_polare_sine(self):
    theta = np.linspace(0, 2 * np.pi, 500)
    num_funzioni = 10
\section{Base delle funzioni sinusoidali}
    r_base = [np.abs(np.sin(theta * i / 2)) for i in range(1, num_funzioni + 1)]
```

Questo mostra come per `i` crescenti si ottengono curve polari con sempre più lobi, che in Csound diventano pattern di inviluppo sempre più complessi.

La conversione finale da coordinate polari a stereo avviene con:

```
; Calcolo delle componenti Mid-Side
kMid = cos(krad)
kSide = sin(krad)

; Applicazione dell'inviluppo al segnale
aMid = kMid * asigEnv
aSide = kSide * asigEnv

; Conversione a Left-Right con matrice di rotazione
aL = (aMid + aSide) / $SQRT2
aR = (aMid - aSide) / $SQRT2
```

2.2.3 Gestione degli Inviluppi Multipli

Il sistema gestisce due livelli di inviluppo che interagiscono moltiplicativamente:

```
; Inviluppo locale dell'evento (derivato dalle armoniche spaziali)
asigLocalEnv = asig * kEnv_local

; Inviluppo di sezione (se presente)
kEnv_section = 1
if i_ifn_section_env > 20 && i_section_duration > 0 then
    k_time_absolute times
    k_time_since_section_start = k_time_absolute - i_section_start_time
    kndx_section = limit(k_time_since_section_start / i_section_duration, 0, 1)
    kEnv_section tablei kndx_section, i_ifn_section_env, 1
endif

; Combinazione degli inviluppi
asigEnvPre = asigLocalEnv * kEnv_section
asigEnv dcblock asigEnvPre
```

L'inviluppo di sezione permette modulazioni globali su tutti gli eventi di una sezione, mentre l'inviluppo locale (potenzialmente derivato dalle armoniche spaziali) definisce la forma del singolo evento.

2.3 Il Sistema di Intonazione Pitagorica

Il sistema di altezze in Gamma si basa su una implementazione personalizzata dell'intonazione pitagorica studiata sugli scritti di Walter Branchi [2], gestita dall'opcode `GenPythagFreqs`:

```
opcode GenPythagFreqs, i, iiii
  iFund, iNumIntervals, iNumOctaves, iTblNum xin
  iTotalLen = iNumIntervals * iNumOctaves
  iFreqs[] init iTotalLen

iOctave = 0
  iBaseIndex = 0

while iOctave < iNumOctaves do
    iFifth = 3/2
    iFreqs[iBaseIndex] = iFund * (2^iOctave)

; Genera la serie di quinte per questa ottava
    indx = 1
    iLastRatio = 1
    while (indx < iNumIntervals) do
      iRatio = iLastRatio * iFifth
      ; Riduci all'ottava␣di␣riferimento
␣␣␣␣␣␣while␣(iRatio␣>=␣2)␣do
␣␣␣␣␣␣␣␣iRatio␣=␣iRatio␣/␣2
␣␣␣␣␣␣od
␣␣␣␣␣␣iFreqs[iBaseIndex␣+␣indx]␣=␣iFund␣*␣iRatio␣*␣(2^iOctave)
␣␣␣␣␣␣iLastRatio␣=␣iRatio
␣␣␣␣␣␣indx␣+=␣1
␣␣␣␣od
```

Il sistema genera una tabella bidimensionale concettuale dove:

- Ogni ottava contiene `iNumIntervals` frequenze (200 nel nostro caso)
- Le frequenze sono generate attraverso iterazioni della quinta perfetta (3/2)
- Ogni quinta che supera l'ottava viene riportata all'interno tramite divisione per 2

Dopo la generazione, le frequenze vengono ordinate all'interno di ogni ottava:

```
; Ordina le frequenze per questa ottava
indx = iBaseIndex
while (indx < (iBaseIndex + iNumIntervals - 1)) do
  indx2 = indx + 1
  while (indx2 < (iBaseIndex + iNumIntervals)) do
    if (iFreqs[indx2] < iFreqs[indx]) then
```

```
      iTemp = iFreqs[indx]
      iFreqs[indx] = iFreqs[indx2]
      iFreqs[indx2] = iTemp
    endif
    indx2 += 1
  od
  indx += 1
od
```

Questo bubble sort garantisce che le frequenze siano accessibili in ordine crescente all'interno di ogni ottava.

2.3.1 Mappatura Ottava-Registro-Ritmo

L'accesso alle frequenze avviene attraverso la funzione `calcFrequenza`:

```
opcode calcFrequenza, i, iii
    i_Ottava, i_Registro, i_RitmoCorrente xin

; Calculate octave register
    i_Indice_Ottava = int(i_Ottava * $INTERVALLI)
    ; Calculate interval offset within the octave
    i_OffsetIntervallo = i_Indice_Ottava + int(((i_Registro * $INTERVALLI) /
      ↪ $REGISTRI))

; Get the frequency from the table using the calculated offset
    i_Freq table max(1, i_OffsetIntervallo + i_RitmoCorrente), gi_Intonazione
    ifreq = min(i_Freq, sr/2-1)
    xout ifreq
endop
```

La formula di indicizzazione `i_OffsetIntervallo + i_RitmoCorrente` crea una relazione diretta tra il valore ritmico e l'altezza selezionata. Questo significa che:

- Ritmi identici in registri diversi producono intervalli correlati
- La sequenza ritmica diventa una sequenza melodica
- Valori ritmici alti tendono verso frequenze più acute all'interno del registro

2.3.2 Implicazioni Compositive

Questa architettura crea una profonda interconnessione tra dimensione temporale, frequenziale e spaziale. Un pattern ritmico [3, 5, 8, 13] definisce:

- Le durate primarie relative degli eventi (durataArmonica/3, durataArmonica/5, etc.) (primarie poiché trasfigurate successivamente da un moltiplicatore di durata).

- Le altezze selezionate dalla tabella pitagorica
- Il numero di suddivisioni spaziali e il pattern di movimento stereofonico
- La forma dell'inviluppo di ampiezza quando si usano le armoniche spaziali

L'uso dell'intonazione pitagorica invece del temperamento equabile aggiunge ulteriore ricchezza armonica: le quinte sono pure (rapporto 3:2), ma questo genera comma pitagorici e intervalli microtonali che colorano il risultato con battimenti multipli di difficile prevedibilità.

2.4 NonlinearFunc: Il Generatore di Ritmi Caotici

L'opcode `NonlinearFunc` rappresenta un sistema per la generazione di sequenze ritmiche che evolvono nel tempo secondo principi deterministici, periodici o caotici. Questo UDO (User Defined Opcode) estende le possibilità compositive oltre i pattern ritmici predefiniti, permettendo l'esplorazione di territori ritmici emergenti.

2.4.1 Struttura e Parametri dell'Opcode

```
opcode NonlinearFunc, i, ippo
  iX, iMode, iMinVal, iMaxVal xin

; Valori di default per min/max se non specificati
  iMinVal = (iMinVal == 0) ? 1 : iMinVal
  iMaxVal = (iMaxVal == 0) ? 35 : iMaxVal

; Assicurati che iX sia entro limiti sensati
  iX = limit(iX, 1, 100)

iPI = 4 * taninv(1.0)
  iTemp = 0
```

L'opcode accetta quattro parametri:

- `iX`: Il valore di input, tipicamente il ritmo precedente nella sequenza
- `iMode`: Selettore della modalità operativa (0-3)
- `iMinVal`: Valore minimo del range di output (default: 1)
- `iMaxVal`: Valore massimo del range di output (default: 35)

La prima operazione importante è la normalizzazione e limitazione dei valori di input per garantire stabilità numerica. Il valore di `iX` viene limitato tra 1 e 100 per evitare overflow o comportamenti indefiniti nelle funzioni matematiche successive.

2.4.2 Modalità 0: Convergente

```
if iMode == 0 then
    ; --- MODALITÀ 0: CONVERGENTE ---
    iR = 2.8
    iTemp = iR * iX * (1 - iX/40)
```

Questa modalità implementa una variante della mappa logistica con comportamento convergente. Il parametro `iR = 2.8` è scelto specificamente per rimanere nella regione stabile del diagramma di biforcazione della mappa logistica, dove il sistema converge verso un punto fisso.

La formula `iR * iX * (1 - iX/40)` differisce dalla classica mappa logistica `r * x * (1 - x)` per il fattore di scala 40. Questo adattamento:

- Permette di lavorare con valori di input nell'intervallo 1-100 invece di 0-1
- Rallenta la convergenza, rendendo l'evoluzione ritmica più graduale
- Crea una traiettoria prevedibile verso un valore stabile

Matematicamente, per `iR = 2.8`, il sistema convergerà verso il punto fisso:

```
x* = 40 * (1 - 1/iR) ≈ 25.71
```

Questo significa che sequenze ritmiche in modalità convergente tenderanno gradualmente verso valori intorno a 26, creando un effetto di stabilizzazione ritmica.

2.4.3 Modalità 1: Periodica

```
elseif iMode == 1 then
    ; --- MODALITÀ 1: PERIODICA ---
    iP1 = sin(iX * iPI/18)
    iP2 = cos(iX * iPI/10)
    iTemp = abs(iP1 * iP2) * 20 + 10
```

La modalità periodica utilizza l'interferenza di due funzioni trigonometriche con periodi incommensurabili per generare pattern complessi ma deterministici.

L'analisi matematica rivela:

- `sin(iX * pi/18)`: periodo di 36 unità
- `cos(iX * pi/10)`: periodo di 20 unità
- Il minimo comune multiplo è 180, creando un super-periodo

Il prodotto `iP1 * iP2` genera un'interferenza costruttiva e distruttiva tra le due onde:

- Quando entrambe le funzioni sono vicine ai loro massimi/minimi, il prodotto è grande
- Quando una è vicina a zero, il prodotto si annulla
- Il valore assoluto garantisce output positivi

La trasformazione finale `abs(iP1 * iP2) * 20 + 10`:

- Scala il range da [0, 1] a [0, 20]
- Aggiunge un offset di 10, risultando in valori tra 10 e 30
- Garantisce che i ritmi generati rimangano in un range musicalmente utile

Questa modalità produce sequenze che si ripetono dopo 180 iterazioni ma con una struttura interna ricca di variazioni locali.

2.4.4 Modalità 2: Caotica Deterministica

```
elseif iMode == 2 then
    ; --- MODALITÀ 2: CAOTICA DETERMINISTICA ---
    iR = 3.99
    iNormX = (iX % 100) / 100
    iNormX = limit(iNormX, 0.01, 0.99)
    iLogistic = iR * iNormX * (1 - iNormX)
    iNoise = random:i(-0.05, 0.05)
    iLogistic = limit(iLogistic + iNoise, 0, 1)
    iRange = iMaxVal - iMinVal + 1
    iTemp = iMinVal + (iLogistic * iRange)
```

Questa modalità implementa la mappa logistica nella sua regione caotica con l'aggiunta di una piccola perturbazione stocastica.

Il parametro `iR = 3.99` posiziona il sistema al limite del caos:

- Per r > 3.57, la mappa logistica entra nel regime caotico
- A r = 3.99, siamo nella regione di caos sviluppato
- Piccole variazioni nell'input producono grandi divergenze nell'output

Il processo di normalizzazione `(iX % 100) / 100`:

- Utilizza l'operatore modulo per mantenere i valori ciclici

- Normalizza nell'intervallo [0, 1] richiesto dalla mappa logistica
- Il limite `[0.01, 0.99]` evita i punti fissi instabili a 0 e 1

L'aggiunta di rumore `random:i(-0.05, 0.05)`:

- Introduce una componente stocastica del 5%
- Previene cicli perfetti che potrebbero emergere anche nel caos deterministico
- Simula le imperfezioni del mondo reale

2.4.5 Modalità 3: Caos Vero (Default)

```
else
    ; --- MODALITÀ 3: CAOS VERO (DEFAULT) ---
    ; 1. Componente deterministica (60%)
    iSeed1 = (iX * 1.3) % 10
    iSeed2 = (iX * 0.7) % 10
    iSeed3 = (iX * 2.5) % 10
    iNonlinear1 = abs(sin(iSeed1 * iPI/5 + iSeed2))
    iNonlinear2 = abs(cos(iSeed2 * iPI/3 + iSeed3))
    iNonlinear3 = abs(tan(iSeed3 * iPI/7 + iSeed1) % 1)
    iDeterministic = (iNonlinear1 + iNonlinear2 + iNonlinear3) / 3
```

La modalità *Caos Vero* rappresenta l'approccio più interessante, combinando molteplici generatori non lineari con componenti stocastiche.

La generazione dei seed utilizza moltiplicatori irrazionali approssimati:

- $1.3 \approx \sqrt{1.69}$
- $0.7 \approx 1/\sqrt{2}$
- $2.5 \approx \sqrt{6.25}$

Questi valori garantiscono che i tre seed evolvano a velocità diverse e incommensurabili, massimizzando la complessità dell'output.

Le tre funzioni non lineari utilizzano:

- `sin` con accoppiamento additivo: sensibile alle fasi relative
- `cos` con accoppiamento additivo: sfasato di $\pi/2$ rispetto a sin
- `tan` con modulo: introduce discontinuità controllate

```
    ; 2. Componente casuale (40%)
    iRandom = random:i(0, 1)

; 3. Combina le componenti
    iMixRatio = 0.6
    iCombined = (iDeterministic * iMixRatio) + (iRandom * (1 - iMixRatio))
```

Il bilanciamento 60/40 tra deterministico e stocastico è calibrato per:

- Mantenere una struttura riconoscibile (componente deterministica)
- Introdurre sufficiente imprevedibilità (componente random)
- Evitare sia la monotonia che il rumore bianco

```
    ; 4. Perturbazione periodica
    iPerturbation = 0
    if (iX % 7 == 0) then
      iPerturbation = random:i(-0.3, 0.3)
    endif

; 5. Mappa al range finale
    iRange = iMaxVal - iMinVal + 1
    iTemp = iMinVal + (iCombined * iRange) + (iPerturbation * iRange)
```

La perturbazione periodica ogni 7 iterazioni:

- Introduce eventi rari ma significativi
- Il numero 7 (primo) evita risonanze con altri periodi nel sistema
- L'ampiezza ±30% può causare salti drammatici nel ritmo

2.4.6 Integrazione con il Sistema Gamma

Nel contesto dello strumento Voce, NonlinearFunc viene chiamato quando la tabella dei ritmi predefiniti si esaurisce:

```
i_RitmoCorrente NonlinearFunc i_Vecchio_Ritmo, i_NonlinearMode
```

Questo crea una transizione fluida da:

1. **Fase deterministica**: Ritmi composti e memorizzati in tabella
2. **Fase generativa**: Ritmi creati algoritmicamente

L'output di NonlinearFunc influenza direttamente:

- **Temporalità**: Attraverso la formula `i_DurataArmonica / i_RitmoCorrente`
- **Altezza**: Il ritmo viene usato come indice nella tabella delle frequenze
- **Spazializzazione**: Determina il parametro iHR per le armoniche spaziali

3 CONFIGURAZIONE E PARAMETRIZZAZIONE

Il sistema Gamma implementa una separazione netta tra logica di sintesi e configurazione dei parametri, permettendo al compositore di modificare profondamente il comportamento del sistema senza toccare il codice Csound. Questo approccio modulare facilita la sperimentazione e l'estensione del sistema.

3.1 Il File tables.yaml

Il file `tables.yaml` rappresenta il cuore configurabile del sistema, definendo tutte le tabelle di forma d'onda e inviluppo utilizzate nella sintesi. La sua struttura gerarchica separa chiaramente gli inviluppi per eventi singoli da quelli per sezioni intere.

Ogni tabella è definita attraverso quattro parametri fondamentali:

```
nome_simbolico:
  number: [numero della f-table in Csound]
  size: [dimensione in campioni]
  gen_routine: [numero della GEN routine]
  parameters: [lista dei parametri per la GEN]
```

Vediamo un esempio concreto:

```
event_envelopes:
  lineare:
    number: 2
    size: 4096
    gen_routine: 6
    parameters: [0.001, 2048, 0.5, 2048, 1]  # Linea retta da 0 a 1
```

Questa definizione genera in Csound:

```
f 2 0 4096 6 0.001 2048 0.5 2048 1
```

Il sistema distingue due categorie di inviluppi con funzioni distinte:

`Event Envelopes` Applicati ai singoli eventi sonori:

```
event_envelopes:
  impulsivo:
    number: 5
    size: 4096
```

```
    gen_routine: 5
    parameters: [0.001, 512, 1, 3584, 0.0001]

lento:
    number: 6
    size: 4096
    gen_routine: 7
    parameters: [0, 3072, 1, 1024, 0]

sostenuto:
    number: 7
    size: 4096
    gen_routine: 7
    parameters: [0, 512, 1, 3072, 1, 512, 0]
```

L'inviluppo `impulsivo` utilizza GEN 5 (segmenti esponenziali) per creare un attacco rapidissimo (512 campioni su 4096, circa 1/8 della durata) seguito da un decadimento esponenziale. Il valore finale di 0.0001 invece di 0 evita discontinuità nell'interpolazione esponenziale.

L'inviluppo `lento` con GEN 7 crea un attacco graduale per 3/4 della durata, ideale per tessiture ambient o crescendi graduali.

`Section Envelopes` Modulano interi gruppi di eventi:

```
section_envelopes:
  crescendo_diminuendo:
    number: 24
    size: 4096
    gen_routine: 7
    parameters: [0, 2048, 1, 2048, 0]

impulso:
    number: 25
    size: 4096
    gen_routine: 6
    parameters: [1, 4096, 0.001]
```

Gli inviluppi di sezione operano su una scala temporale maggiore. Il numero di tabella parte da 20 per convenzione, distinguendoli chiaramente dagli inviluppi evento nel codice Csound:

```
if i_ifn_section_env > 20 && i_section_duration > 0 then
    ; Applica inviluppo di sezione
endif
```

3.2 Sistema di Macro e Costanti Globali

Il template CSD di Gamma definisce un sistema di macro che parametrizza l'intero spazio frequenziale:

```
#define FONDAMENTALE #32#
#define OTTAVE #10#
#define INTERVALLI #200#
```

```
4 #define REGISTRI #50#
```

Le macro `OTTAVE`, `INTERVALLI` e `REGISTRI` definiscono la risoluzione del sistema di intonazione:

- `OTTAVE` (10): Copre l'intero range udibile da 32 Hz a ~32 kHz
- `INTERVALLI` (200): Numero di divisioni per ottava nel sistema pitagorico
- `REGISTRI` (50): Suddivisioni macro all'interno di ogni ottava

La relazione tra questi parametri determina la granularità frequenziale:

```
1 Totale frequenze = OTTAVE * INTERVALLI = 2000
2 Risoluzione per registro = INTERVALLI / REGISTRI = 4 intervalli
```

Il file `tables.yaml` viene letto dal generatore Python che:

1. Carica le configurazioni all'inizializzazione
2. Genera automaticamente gli f-statement nel CSD
3. Mantiene mappe nome→numero per riferimenti simbolici

Questo permette di riferirsi agli inviluppi per nome nel YAML compositivo:

```
1 inviluppo_attacco: { value: 'impulsivo' }
```

Invece di numeri magici:

```
1 inviluppo_attacco: { value: 5 }  # Meno leggibile e manutenibile
```

4 INTRODUZIONE E ARCHITETTURA GENERALE

Il `generative_composerYaml2.py` è un sistema di composizione algoritmica che traduce una descrizione astratta di una struttura musicale, definita in formato YAML, in un file audio (WAV). Lo fa generando uno score per il software di sintesi sonora Csound.

L'architettura del programma è basata su tre componenti principali e un'esecuzione a fasi:

1. **`GenerativeComposer`**: La classe principale che contiene la logica per interpretare la partitura YAML, generare i parametri stocastici degli eventi sonori e creare i file di score `.csd` per Csound.

2. **`CompositionDebugger`**: Una classe di utilità dedicata esclusivamente alla creazione di una visualizzazione grafica (in formato PDF) della composizione generata, simile a un *piano roll* arricchito con informazioni sulle tendenze parametriche.
3. **`TimeScheduler`**: Una classe specializzata nella generazione di sequenze temporali (gli *onset*, o istanti di inizio) degli eventi, secondo diversi modelli (lineare, accelerando, ritardando, etc.).

Il processo, orchestrato nel blocco `if __name__ == "__main__":`, non è monolitico ma suddiviso in fasi distinte e sequenziali, che permettono di separare la generazione, il rendering e la visualizzazione.

4.1 Fase di Input: Caricamento della Struttura della Composizione

Il punto di partenza è un file YAML. Il programma supporta la definizione di composizioni complesse, articolate in più parti, utilizzando la sintassi multi-documento di YAML (documenti separati da `---`).

La funzione `load_all_compositions_from_yaml` si occupa di questo compito:

```
def load_all_compositions_from_yaml(file_path):
    """
    Carica una o più composizioni da un singolo file YAML.
    I documenti multipli devono essere separati da '---'.
    Restituisce una lista di strutture di composizione.
    """
    print(f"Caricamento partiture dal file multi-documento: {file_path}")
    composizioni = []
    try:
        with open(file_path, 'r') as f:
            docs = list(yaml.safe_load_all(f))
        for i, composition in enumerate(docs):
            if composition is None: continue
            composizioni.append(composition)
        return composizioni
```

Ogni documento YAML caricato rappresenta una *Parte* della composizione. Ogni parte è una lista di *Sezioni*, e ogni sezione può contenere uno o più *Layer*. Questa struttura gerarchica (Parte -> Sezione -> Layer -> Evento) è il modello concettuale su cui si basa tutta la logica successiva.

In aggiunta, viene caricato un file `tables.yaml` che definisce le caratteristiche degli inviluppi (es. attacco, rilascio) che verranno usati da Csound.

4.2 Il Nucleo Generativo: Dal Concetto ai Parametri

Il cuore del sistema risiede nella classe `GenerativeComposer` e nella sua capacità di trasformare le *maschere* parametriche definite nel YAML in valori numerici concreti per ogni

evento sonoro.

La generazione avviene all'interno di un *layer*. Un layer può essere:

- **Statico**: Definito da uno `stato_unico`. Tutti gli eventi generati in questo layer attingeranno da un'unica maschera di parametri.
- **Dinamico**: Definito da uno `stato_iniziale` e uno `stato_finale`. I parametri degli eventi evolvono nel tempo, interpolando tra queste due maschere.

La funzione `_process_layer` gestisce un singolo layer. I suoi passaggi chiave sono:

1. **Calcolo del Timing**: Determina la durata effettiva del layer basandosi sul `lifespan` (una finestra temporale relativa alla sezione, es. `[0.0, 0.5]` per la prima metà).
2. **Generazione degli Onset**: Utilizza `TimeScheduler` per calcolare gli istanti di attivazione dei cluster di eventi all'interno della durata del layer.
3. **Generazione degli Eventi**: Per ogni onset, determina la maschera parametrica (statica o interpolata) e genera un *cluster* di eventi sonori.

4.2.1 Generazione Stocastica dei Parametri (`_generate_params_from_mask`)

Questa funzione è il motore stocastico. Prende una *maschera* (un dizionario che descrive un parametro) e produce un valore numerico. Supporta diversi tipi di generazione:

- **Distribuzione Uniforme**: Se la maschera contiene una chiave `range`.

```
elif 'range' in p_mask:
    min_val, max_val = p_mask['range']
\section{...}
    val = random.uniform(min_val, max_val)
```

- **Distribuzione Normale**: Se la maschera contiene `mean` e `std`.

```
if 'mean' in p_mask and 'std' in p_mask:
    mean = p_mask['mean']
    std = p_mask['std']
    val = np.random.normal(loc=mean, scale=std)
```

- **Scelta Pesata**: Se la maschera contiene `choices` ed opzionalmente `weights`.

```
elif 'choices' in p_mask:
    val = random.choices(p_mask['choices'], weights=p_mask.get('weights'), k=1)
      ↪ [0]
```

Questa logica viene applicata a tutti i parametri (altezza, durata, etc.), rendendo il sistema flessibile.

4.2.2 Interpolazione dei Parametri (`_interpolate_mask`)

Per i layer dinamici, questa funzione calcola una maschera intermedia tra `start_mask` e `end_mask` in base a un valore di `progress` (da 0 a 1). L'interpolazione è intelligente e si adatta al tipo di parametro:

- I parametri numerici (come `mean`, `std`, `range`) vengono interpolati linearmente.
- I parametri basati su scelte (`choices`) subiscono un *cross-fade* dei loro pesi (`weights`), creando una transizione probabilistica graduale da un set di scelte a un altro.

```
\section{Esempio di interpolazione di un range}
if 'range' in s_mask:
    s_min, s_max = s_mask['range']
    e_min, e_max = e_mask.get('range', s_mask['range'])
    i_min = s_min + (e_min - s_min) * shaped_progress
    i_max = s_max + (e_max - s_max) * shaped_progress
    interp_mask[key]['range'] = [i_min, i_max]
```

4.2.3 Generazione dello Score Csound (`generate_csd`)

Una volta generata la sequenza completa di eventi, la funzione `generate_csd` assembla il file `.csd`. Non scrive codice Csound complesso, ma piuttosto popola un template.

1. **Tabelle Dinamiche (`f-statements`)**: Crea le tabelle per i pattern ritmici e gli inviluppi.
2. **Eventi (`i-statements`)**: Itera su ogni evento generato e scrive una riga di score (`i "Voce" ...`) con tutti i parametri calcolati.

```
\section{Frammento della riga di score generata}
score_lines += (f'i "Voce"\t{event_time:.4f}\t{p["durata_totale"]:.3f}\t'
                f'{p["ritmi_tab_num"]}\t{p["durata_armonica"]:.3f}\t\t{p["
                  ↪ dynamic_index"]:.6f}\t'
\section{... altri parametri ...}
              )
```

Questo file `.csd` è un output intermedio, pronto per essere processato da Csound per generare un file audio.

4.3 L'Orchestrazione del Rendering (Blocco `__main__`*)*

L'approccio del compositore al rendering è granulare e mira a ottimizzare i tempi di lavoro, specialmente su composizioni complesse. Questo avviene attraverso una sequenza di fasi ben definita.

4.3.1 Fase 1: `plan_render_jobs`

Questa funzione analizza l'intera struttura della partitura e crea un *piano di lavoro*. Non esegue alcun rendering, ma definisce *cosa-deve essere renderizzato. Per ogni layer che necessita di rendering, crea un *job*, ovvero un dizionario contenente:

- La definizione del layer e della sezione a cui appartiene.
- I percorsi per i file `.csd` e `.wav` di output per quel singolo layer.
- Il tempo di inizio assoluto della sezione, calcolato tenendo conto del parametro `offset_inizio`.

4.3.2 Fase 2: `execute_layer_rendering_and_collect_data`

Questa fase esegue i *job* di rendering dei layer.

1. Per ogni job, invoca la logica di `GenerativeComposer` per generare gli eventi solo per quel layer.
2. Genera un file `.csd` specifico per il layer.
3. Lancia un processo Csound (`subprocess.Popen`) per renderizzare il `.csd` del layer in un file `.wav`.
4. **Crucialmente**, raccoglie tutti i dati degli eventi generati in una struttura dati (`plot_data`). Questi dati sono essenziali per la visualizzazione.

Questo approccio permette di renderizzare solo i layer modificati se si utilizza la `veteranMode`, una modalità che salta il rendering dei layer non contrassegnati come `veteranMode: True`.

4.3.3 Fase 3: `generate_composition_plot` *e la Cache di Visualizzazione*

Questa fase si occupa della visualizzazione. La sua caratteristica principale è l'uso di un file cache, `visual_cache.json`:

1. **Lettura della Cache**: Carica i dati di visualizzazione da esecuzioni precedenti, se disponibili.
2. **Merge**: Se sono stati generati nuovi dati (da `execute_layer_rendering` **...**), questi vengono uniti alla cache, sostituendo i dati vecchi per i layer che sono stati ri-renderizzati.
3. **Plotting**: Usa la classe `CompositionDebugger` per creare un PDF multi-pagina che mostra: - Gli eventi sonori come rettangoli. - Le *maschere di tendenza* (le buste grigie/arancioni) che mostrano l'evoluzione dei range parametrici. - L'evoluzione delle dinamiche. - Marcatori per sezioni e layer.
4. **Scrittura della Cache**: Salva lo stato aggiornato dei dati di visualizzazione nel file JSON. Questo garantisce che, alla prossima esecuzione in `veteranMode`, il grafico mostri comunque l'intera composizione (parti vecchie e nuove). La funzione `_sanitize_data_for_json` è un helper fondamentale qui, poiché converte tipi di dati specifici di NumPy e `pathlib` in formati compatibili con JSON.

4.3.4 Fase 4 e 5: Assemblaggio

Il rendering finale non avviene generando un unico, enorme file CSD. Avviene invece tramite un processo di assemblaggio gerarchico:

1. `execute_section_assembly`: Per ogni sezione, genera un CSD *assembler*. Questo CSD non produce suono, ma si limita a leggere e mixare i file `.wav` dei singoli layer (generati nella Fase 2) per creare un unico file `.wav` per l'intera sezione.
2. `execute_final_assembly`: Genera un ultimo CSD *assembler* che prende i file `.wav` di tutte le sezioni e li posiziona in sequenza (rispettando gli `offset_inizio`) per creare il file `.wav` finale e completo della composizione.

Questo approccio a *render per layer, poi assembla* ha il vantaggio di essere più gestibile e di non richiedere la rigenerazione dell'intera composizione per piccole modifiche.

Il `generative_composerYaml2.py` implementa un flusso di lavoro completo e disaccoppiato per la composizione algoritmica. Le sue caratteristiche tecniche salienti sono:

- **Configurazione Esterna (YAML)**: Offre un'interfaccia di alto livello per la descrizione musicale, separando la logica del codice dai dati della composizione.

- **Generazione Stocastica Multi-modello**: Fornisce un set flessibile di strumenti per definire il comportamento dei parametri sonori.
- **Rendering Granulare e a Fasi**: Scompone il problema del rendering in sotto-problemi più piccoli (layer, sezioni), ottimizzando il processo di lavoro iterativo.
- **Caching della Visualizzazione**: Garantisce che il feedback visivo sia sempre coerente e completo, anche quando si lavora solo su parti della composizione.
- **Assemblaggio Gerarchico**: Utilizza Csound non solo per la sintesi ma anche come uno strumento di montaggio audio per assemblare i componenti finali.

5 DALL'INTENZIONE ALLA NOTA: MASCHERE DI TENDENZA E GENERAZIONE PARAMETRICA

La generazione parametrica è il motore alchemico di Gamma, il processo centrale attraverso cui le intenzioni del compositore, espresse come maschere di tendenza nel file YAML, vengono trasformate in valori numerici concreti per ogni singolo evento sonoro. Analizzeremo come il sistema traduce l'astrazione in suono, con particolare attenzione alle tecniche di generazione, interpolazione e gestione della coerenza strutturale.

5.1 Il Motore Generativo: `_generate_params_from_mask()`

Il metodo `_generate_params_from_mask()` è il punto di convergenza tra l'astrazione compositiva e la concretezza numerica. Implementa la logica che trasforma una singola maschera di tendenza nei parametri specifici per un evento sonoro, gestendo una varietà di strategie generative per rispondere a diverse necessità espressive.

5.1.1 Architettura e Gestione Specializzata

Il processo non è monolitico. Il metodo prima isola un insieme di chiavi che richiedono una gestione specializzata, poiché non rappresentano parametri diretti o necessitano di logiche di trasformazione complesse.

```
SKIPPED_KEYS = {
    'choices', 'weights', 'distribution',  # Metadati per la generazione
    'dynamic_index', 'dinamica',           # Logica di dinamica complessa
    'nonlinear_mode', 'senso_movimento',   # Parametri di controllo per Csound
    'inviluppo_attacco', 'tipo_ritmi',      # Richiedono traduzione o
      ↪ generazione complessa
    'densita_cluster'
}
```

Questa separazione permette a un loop generico di gestire i parametri puramente numerici, mentre logiche dedicate si occupano di tradurre concetti come `'dinamica': 'f'` nell'in-

dice numerico richiesto da Csound o di generare intere sequenze ritmiche da una categoria come `'medi'`.

5.1.2 Le Modalità di Generazione

Il cuore del metodo itera sui parametri, applicando la modalità di generazione più appropriata in base alla struttura della maschera.

```
for key, p_mask in mask.items():
    if key in SKIPPED_KEYS: continue
\section{1. Distribuzione Normale (Gaussiana)}
    if 'mean' in p_mask and 'std' in p_mask:
        val = np.random.normal(loc=p_mask['mean'], scale=p_mask['std'])
\section{2. Distribuzione Uniforme (Range)}
    elif 'range' in p_mask:
        min_val, max_val = p_mask['range']
        val = random.randint(min_val, max_val) if isinstance(min_val, int) else
          ↪ random.uniform(min_val, max_val)
\section{3. Scelta Discreta Pesata}
    elif 'choices' in p_mask:
        val = random.choices(p_mask['choices'], weights=p_mask.get('weights'), k
          ↪ =1)[0]
\section{4. Valore Fisso}
    elif 'value' in p_mask:
        val = p_mask['value']
```

Questo toolkit di quattro modalità offre al compositore un controllo granulare sul grado di determinismo e casualità:

1. Distribuzione Normale : Ideale per creare una *massa* sonora attorno a un centro tonale o timbrico. Un'ottava definita come `{mean: 5, std: 0.5}` tenderà a rimanere nell'ottava 5, ma con occasionali e naturali *fughe* verso le ottave vicine. La deviazione standard diventa un parametro espressivo che controlla la *disciplina* del materiale.
2. Range Uniforme `range: [1, 10]`: Utile quando tutti i valori in un intervallo sono ugualmente possibili.
3. Scelta Pesata : Permette di definire il *colore* statistico di una sezione. Una dinamica specificata come `{choices: ['p', 'mf', 'f'], weights: [0.6, 0.3, 0.1]}` assicura una predominanza di eventi piano, pur mantenendo la varietà.
4. Valore Fisso : Garantisce il determinismo assoluto, essenziale per parametri strutturali come il senso di movimento spaziale.

5.1.3 La Logica Gerarchica dei Ritmi

La generazione dei ritmi è un esempio emblematico di come il sistema supporti molteplici livelli di astrazione, dal controllo totale alla delega generativa.

```
if 'explicit_values' in rhythm_mask:
\section{Modalità 1: Controllo totale con una lista esplicita}
    params['ritmi'] = rhythm_mask['explicit_values']
elif 'choices' in rhythm_mask:
    choice = random.choices(rhythm_mask['choices'], ...)[0]
    if isinstance(choice, list):
\section{Modalità 2: Scelta tra pattern pre-composti}
        params['ritmi'] = choice
    else:
\section{Modalità 3: Astrazione massima tramite categorie ('piccoli', 'medi'...)
  ↪ }
        params['ritmi'] = self._generate_rhythm_pattern(choice)
```

Questa architettura permette al compositore di scegliere il livello di dettaglio più consono: specificare un pattern esatto, scegliere da una libreria di pattern, o semplicemente indicare una *qualità* ritmica desiderata.

5.2 L'Evoluzione nel Tempo: Interpolazione delle Maschere

Se la generazione da una singola maschera crea eventi statici, l'interpolazione tra due maschere (`stato_iniziale` e `stato_finale`) dà vita a processi dinamici e trasformativi. Il metodo `_interpolate_mask()` implementa questa logica.

5.2.1 Gestione delle Transizioni Asimmetriche

Un problema chiave nell'interpolazione è come gestire parametri che compaiono solo nello stato finale. Gamma adotta una strategia di *riempimento* che ne aumenta la flessibilità.

```
all_keys = set(start_mask.keys()) | set(end_mask.keys())
for key in all_keys:
    s_mask = start_mask.get(key)
    e_mask = end_mask.get(key)

if s_mask is None: s_mask = e_mask
    if e_mask is None: e_mask = s_mask
```

Se un parametro è definito solo alla fine, il sistema assume che fosse presente fin dall'inizio con lo stesso valore finale. Questo permette di *introdurre* un nuovo processo (es. un glissando) senza doverne specificare un valore nullo all'inizio, semplificando la scrittura delle partiture YAML.

5.2.2 Strategie di Interpolazione Differenziate

L'interpolazione si adatta al tipo di parametro, producendo transizioni musicalmente significative:

- Parametri Numerici (Range, Mean/Std) : I loro valori vengono interpolati linearmente. Questo permette di creare effetti come un *restringimento* del campo sonoro (interpolando verso un range più piccolo) o una *focalizzazione* (interpolando verso una deviazione standard minore).

- Scelte Discrete (Choices) : Il sistema tenta un cross-fade probabilistico . Se le scelte sono le stesse, i loro pesi (`weights`) vengono interpolati. Questo rea una transizione graduale nella probabilità di occorrenza, ad esempio passando da una predominanza di dinamiche piano a una di forte. Se le scelte sono diverse, il sistema effettua una transizione a scalino a metà del percorso.

È importante notare che questo processo di interpolazione non solo guida la generazione degli eventi sonori, ma fornisce anche i dati per la visualizzazione grafica. Le *maschere di tendenza* visibili nel PDF generato da `CompositionDebugger` sono la rappresentazione visiva diretta dei valori interpolati in ogni punto del tempo. Questo crea una coerenza totale tra ciò che il compositore specifica, ciò che il sistema visualizza e ciò che l'orchestra Csound suona.

5.3 Il Sistema Gerarchico del Glissando

Il glissando in Gamma non è una semplice transizione di frequenza, ma un sistema gerarchico che illustra l'approccio progettuale del compositore: fornire opzioni potenti con priorità chiare.

1. Modalità Offset (Priorità Massima): Specifica un intervallo di glissando relativo alla nota di partenza (es. `offset_ottava: 2`). È ideale per creare pattern di movimento che mantengono la loro coerenza intervallare a diverse altezze.
2. Modalità Assoluta (Priorità Media) : Specifica una destinazione fissa (es. `ottava_arrivo: 8`). È utile per creare convergenze armoniche, dove più voci, partendo da punti diversi, si dirigono verso un'unica regione tonale.
3. Default (Nessun Glissando) : In assenza di specifiche, la frequenza rimane statica.

Questa gerarchia viene risolta in `_generate_params_from_mask()`, che calcola i parametri `ottava_arrivo` e `registro_arrivo` finali. Questi vengono poi passati a Csound, dove la funzione `calcFrequenza` è chiamata due volte, una per la frequenza di partenza e una per quella di arrivo, garantendo che entrambe rispettino la logica dell'intonazione pitagorica del sistema.

6 SCOLPIRE IL TEMPO: MODELLI TEMPORALI E MICRO-RITMICA

La classe `TimeScheduler` incapsula la logica per la distribuzione temporale, offrendo un toolkit di modelli che corrispondono a gesti musicali archetipici. La scelta di isolare questa

funzionalità in una classe dedicata sottolinea come il tempo musicale non sia un semplice parametro, ma una dimensione fondamentale che richiede un trattamento specializzato.

Il metodo `generate_onsets` è il cuore di questa classe. La sua architettura è elegante e flessibile: parte sempre da una progressione lineare di base, che viene poi *deformata* o *rimappata* secondo il modello scelto nel file YAML.

```
def generate_onsets(self, model, duration, num_events):
    base_progress = np.linspace(0, 1, num_events, endpoint=False)
    final_progress = np.zeros_like(base_progress)
    model_type = model.get('type', 'linear')
    return final_progress * duration
```

Questa architettura a due fasi (generazione di una progressione normalizzata e successiva trasformazione) permette di definire gesti temporali indipendentemente dalla durata effettiva, rendendoli riutilizzabili e scalabili.

6.0.1 I Modelli Archetipici

Lineare (`type: linear`)

Il modello di default, che distribuisce gli eventi in modo equidistante. Sebbene semplice, è fondamentale per creare pulsazioni regolari, ostinati o griglie ritmiche stabili su cui altri layer possono costruire complessità.

Ritardando (`type: ritardando`)

```
elif model_type == 'ritardando':
    shape = model.get('shape', 2.0)
    final_progress = base_progress ** shape
```

Applicando una funzione di potenza con esponente maggiore di 1, la curva di progressione si *piega*, concentrando gli eventi all'inizio e diradandoli verso la fine. Il parametro `shape` controlla l'intensità del gesto: un valore più alto crea un ritardando pronunciato.

Accelerando (`type: accelerando`)

```
elif model_type == 'accelerando':
    shape = model.get('shape', 2.0)
    final_progress = 1 - (1 - base_progress) ** shape
```

La formula inverte la progressione, applica la potenza e la inverte nuovamente. **Stocastico (`type: stochastic`)**

```
elif model_type == 'stochastic':
    final_progress = np.sort(np.random.rand(num_events))
```

Il metodo genera un set di istanti casuali e poi li ordina. Questo garantisce che, pur essendo irregolari, gli eventi mantengano una progressione temporale in avanti. È un modello efficace per rompere la rigidità della griglia metrica.

6.0.2 Il Modello `breakpoint`*: Curve Temporali su Misura*

Il modello più potente e flessibile è `breakpoint`. Permette al compositore di *disegnare* una curva di distribuzione temporale definendo una serie di punti di controllo. Ogni segmento tra due punti può avere la propria curvatura, consentendo la creazione di profili complessi.

Consideriamo un esempio:

```
timing_model:
  type: breakpoint
  points:
    - [0.0, 0.0]        # Inizio
    - [0.3, 0.7, 0.5]   # Il 70% degli eventi avviene nel primo 30% del tempo (curva concava, ease-out)
    - [1.0, 1.0, 3.0]   # Il restante 30% degli eventi si distribuisce nel 70% del tempo rimanente (curva convessa, ease-in)
```

Questo YAML descrive un gesto di *esplosione e diradamento*: un'alta densità di eventi all'inizio, seguita da una lunga coda rarefatta.

L'implementazione gestisce questa complessità in modo modulare:

```
time_in_segment = (segment_times - t_start) / (t_end - t_start)
shaped_time = time_in_segment ** shape
interpolated_values = v_start + (v_end - v_start) * shaped_time
final_progress[segment_mask] = interpolated_values
```

La logica chiave è la normalizzazione del tempo *all'interno di ogni segmento*. Questo permette di applicare una curva di `shape` locale senza che questa influenzi gli altri segmenti, rendendo il sistema potente e intuitivo.

7 SINTASSI E SEMANTICA COMPOSITIVA

In Gamma, la partitura tradizionale lascia il posto a un documento YAML, un formato che diventa un vero e proprio linguaggio per la composizione algoritmica. La grammatica di questo linguaggio si fonda su una gerarchia chiara e intuitiva, che va dalla macro-struttura della composizione al dettaglio del singolo layer sonoro. Al livello più alto troviamo le sezioni, definite da una durata e da parametri strutturali come il `ratio_temporale` o un inviluppo d'ampiezza globale. Ogni sezione funge da contenitore temporale e contestuale per i suoi layer, che rappresentano i veri flussi sonori. È a livello del layer che si definisce l'identità musicale: la densità degli eventi, la loro distribuzione nel tempo tramite un `timing_model` e, soprattutto, le maschere di tendenza che ne governano i parametri.

Questa organizzazione non è casuale, ma guida il compositore a pensare in termini di struttura e tessitura. I parametri di sezione definiscono il *contenitore*, mentre i parametri di layer definiscono il *contenuto*. Il sistema, inoltre, è progettato per essere conciso. Grazie a un modello di eredità implicita, molti parametri assumono valori di default sensati, permettendo al compositore di concentrarsi solo sugli aspetti che desidera modificare. Una semplice maschera come dinamica: 'mf' viene automaticamente espansa dal sistema nella sua forma più strutturata (`{value: 'mf'}`), mantenendo il file di partitura pulito e leggibile.

All'interno di questa struttura, la sintassi di Gamma offre strumenti sofisticati per scolpire la forma nel tempo. Il parametro lifespan, ad esempio, permette di orchestrare entrate e uscite scaglionate dei layer, definendo la loro finestra di attività come una porzione relativa della durata della sezione. Un layer con `lifespan: [0.2, 0.8]` esisterà solo nella parte centrale della sua sezione, permettendo la creazione di forme ad arco e sovrapposizioni complesse che sarebbero macchinose da specificare in modo sequenziale.

La gestione dei confini temporali è ulteriormente raffinata da due meccanismi complementari: il `safety_buffer` e il `leeway`. Il primo è una misura preventiva che accorcia leggermente la finestra di generazione per garantire che nessun evento *sbordi* accidentalmente nella sezione successiva. Il secondo, il `leeway`, agisce come una valvola di sfogo, concedendo agli ultimi eventi di un layer un piccolo margine di tempo extra per concludersi in modo naturale, evitando troncamenti bruschi. Insieme, questi strumenti offrono un controllo rigoroso ma flessibile sui punti di transizione, un aspetto critico nell'assemblaggio di forme musicali estese.

Infine, la sintassi supporta un flusso di lavoro pratico e iterativo. Modalità speciali come `solo: true` permettono di isolare un singolo layer per la messa a punto, mentre il `veteranMode` ottimizza drasticamente i tempi di rendering riutilizzando i file audio dei layer che non sono stati modificati. Queste non sono funzionalità di sintesi, ma strumenti di orchestrazione che rendono il processo compositivo più agile e interattivo.

In definitiva, il linguaggio YAML in Gamma trascende la sua funzione di semplice formato dati. Esso diventa un medium compositivo.

8 CONCLUSIONE

L'analisi ha dimostrato come Gamma realizzi con successo il suo paradigma centrale: quello delle maschere di tendenza. Il compositore definisce i confini, le probabilità, le traiettorie, e il sistema esplora lo spazio creativo così delineato.

Tuttavia l'analisi ha evidenziato anche le sfide intrinseche di questo approccio. La sintassi YAML, pur essendo leggibile, può diventare complessa e verbosa quando si definiscono evoluzioni parametriche sofisticate. Inoltre, il tempo di rendering di Csound rimane il principale collo di bottiglia, suggerendo che per un lavoro più agile potrebbero essere esplorate soluzioni di caching più avanzate o motori di sintesi alternativi per le anteprime.

Queste considerazioni non sminuiscono i risultati, ma anzi tracciano una rotta per il futuro.

Il percorso da Gamma a un ipotetico sistema *Delta* emerge quasi naturalmente dall'architettura esistente. Se Gamma è un sistema dove i layer operano in un elegante isolamento, Delta potrebbe esplorare l'interazione tra gli elementi all'interno dei layer, trasformando la composizione da un insieme di processi paralleli a un vero e proprio sistema di agenti sonori che si ascoltano e si influenzano a vicenda. L'introduzione di una memoria a lungo termine e di metriche di valutazione del materiale generato potrebbe trasformare il sistema da un esecutore di istruzioni a un agente capace di auto-organizzazione.

Riferimenti bibliografici

[1] International Organization for Standardization. Acoustics – normal equal-loudness-level contours. Standard ISO 226:2003, International Organization for Standardization, Geneva, Switzerland, Aug 2003. Orig: International Organization for Standardization, ”A...

[2] I numeri della musica. Roma W. Branchi, 1987. Orig: W. Branchi, I numeri della musica. Roma, Italy: Ed...