

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 sul software di composizione assistita scritto 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 su 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 deterministico controllato.

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 che commistioni tratti caotici e adattivi. 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 - questione che in un sistema chiuso è per definizione gestibile - ma nella complessità intrinseca di progettare e gestire le relazioni interne di uno strumento compositivo senza avere l'esperienza diretta di *suonarlo* e di comporci. È come tentare di costruire uno strumento musicale complesso senza poterlo testare durante la costruzione: la mancanza di un feedback esperienziale rende difficile calibrare le relazioni tra i parametri, prevedere i comportamenti emergenti, e soprattutto sviluppare un'intuizione compositiva per il sistema.

È in questo contesto che nasce *Gamma*, non come ripiego o semplificazione, ma come banco di prova necessario. *Gamma* permette di esplorare in modo più deterministico e controllato le stesse tecniche e strutture che poi verranno integrate nel sistema più complesso di *Delta*. 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 la gamma di possibilità esperibili dello strumento *Delta*. Se *Delta* è la foce dove tutti i flussi compositivi convergono in complessità caotica e adattiva, *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. In fisica, i raggi gamma rappresentano una forma di radiazione elettromagnetica ad alta energia e frequenza, caratterizzata da comportamenti sia ondulatori che corpuscolari. Questa dualità rispecchia perfettamente la natura del sistema compositivo sviluppato, che oscilla costantemente tra determinismo e stocasticità, tra controllo e casualità.

1.2 *Dalle Note alle Nuvole: Un'Eredità Compositiva*

Gamma si inserisce nella tradizione della composizione stocastica e generativa, ispirandosi liberamente ai metodi di lavoro sviluppati da pionieri come Iannis Xenakis e Barry Truax. L'approccio delle *maschere di tendenza* che caratterizza il sistema non è un'invenzione ex novo, ma piuttosto 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 sintesi 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, adattandoli alle esigenze specifiche del progetto Delta. 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 significativo, preparando il terreno per l'evoluzione verso il sistema adattivo previsto.

L'uso delle maschere di tendenza in Gamma permette al compositore 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:** Identificare punti di forza e limitazioni di Gamma, 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 campione audio (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:

```
1 instr Voce
2   ; -----
3   ; 1. INIZIALIZZAZIONE E ACQUISIZIONE PARAMETRI
4   ; -----
5   i_Cattacco      = p2           ; Tempo di attacco del comportamento
6   i_Durata        = p3           ; Durata complessiva
7   i_RitmiTab      = p4           ; Tabella dei ritmi
8   i_DurataArmonica = p5           ; Durata armonica di riferimento
9   i_DynamicIndex  = p6
10  i_Ottava        = p7
11  i_Registro      = p8
12  i_ottava_arrivo = p9
13  i_registro_arrivo = p10
14  i_PosTab        = p11          ; Tabella delle posizioni
15  i_IdComp        = p12          ; ID del comportamento
16  i_NonlinearMode  = (p13 == 0 ? 3 : p13)
17  i_SensoMovimento = (p14 == 0 ? 1 : p14)
18  i_ifnAttacco    = (p15 == 0 ? 10 : p15)
19  i_ifn_section_env = p16
20  i_section_start_time = p17
21  i_duration_leeway = p19
22  i_section_duration = p18 + p19
23  i_section_end     = i_section_start_time + i_section_duration
24  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:

```

1  if i_EventAttack + i_EventDuration > i_section_end then
2      if iSafetyBuffer == 1 then
3          i_EventDuration = i_section_end - i_EventAttack + random:i(0,
4              ↪ i_duration_leeway)
5      endif
6  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 mantenga 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:

```

1  i_EventIdx = 0
2  i_whileTime = 0
3
4  while i_whileTime < i_Durata do
5      ; ----- 3.1 GESTIONE RITMI -----
6      if i_EventIdx < i_LenRitmiTab then
7          i_RitmoCorrente tab_i i_EventIdx, i_TempRitmiTab
8          if i_RitmoCorrente == 0 then
9              goto generateNewRhythm
10         endif
11         i_Vecchio_Ritmo = (i_EventIdx == 0) ? 1 : tab_i(i_EventIdx - 1,
12             ↪ i_TempRitmiTab)
13     else
14         generateNewRhythm:
15         i_Vecchio_Ritmo tab_i i_EventIdx - 1, i_TempRitmiTab
16         i_RitmoCorrente NonlinearFunc i_Vecchio_Ritmo, i_NonlinearMode
17         tabw_i i_RitmoCorrente, i_EventIdx, i_TempRitmiTab
18     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:

```
1 if i_EventIdx == 0 then
2     i_EventAttack = i_CAttacco
3 else
4     i_RitmoNormalizzato = 1 / i_Vecchio_Ritmo
5     i_PreviousAttack tab_i gi_Index - 1, gi_eve_attacco
6     i_EventAttack = i_DurataArmonica * i_RitmoNormalizzato + i_PreviousAttack
7 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:

```
1 i_LenRitmiTab = ftlen(i_RitmiTab)
2 i_TempRitmiTab ftgen 0, 0, i_LenRitmiTab + 10000, -2, 0
3
4 ; Copia i ritmi iniziali nella tabella temporanea
5 i_IndexCopy = 0
6 while i_IndexCopy < i_LenRitmiTab do
7     i_ValRitmo tab_i i_IndexCopy, i_RitmiTab
8     tabw_i i_ValRitmo, i_IndexCopy, i_TempRitmiTab
9     i_IndexCopy += 1
10 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`:

```
1 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 ISO 226:2003. 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	<code>giIsoFreqs</code>	<code>ftgen 0, 0, 32, -2, 20, 25, 31.5, 40, 50, 63, 80, 100, 125, 160, 200,</code> ↪ 250, 315, 400, 500, 630, 800, 1000, 1250, 1600, 2000, 2500, 3150, 4000, ↪ 5000, 6300, 8000, 10000, 12500
2	<code>giAf</code>	<code>ftgen 0, 0, 32, -2, 0.532, 0.506, 0.480, 0.455, 0.432, 0.409, 0.387,</code> ↪ 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	<code>giLu</code>	<code>ftgen 0, 0, 32, -2, -31.6, -27.2, -23.0, -19.1, -15.9, -13.0, -10.3,</code> ↪ -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	<code>giTf</code>	<code>ftgen 0, 0, 32, -2, 78.5, 68.7, 59.5, 51.1, 44.0, 37.5, 31.5, 26.5,</code> ↪ 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:

```
1 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:

```
1 opcode GetIsoAmp, i, ii
2   ifrequency, iDynamicIndex xin
3   iSafeFrequency = limit(ifrequency, 20, 12500)
4
5 ; 1. Recupera i parametri di base per la dinamica data
6   iPhonLevel, iDbfsRef1kHz GetDynamicParams iDynamicIndex
7
8 ; 2. Calcola il dB SPL target per la frequenza e il livello phon dati
9   iDbSplTarget   PhonToSpl_i   iPhonLevel, iSafeFrequency
10
11 ; 3. Il dB SPL di riferimento a 1kHz è per definizione uguale al livello Phon
12   iDbSplRef1kHz   =               iPhonLevel
13
14 ; 4. Calcola l'offset di compensazione
15   iFrequencyOffset = iDbSplTarget - iDbSplRef1kHz
16
17 ; 5. Applica l'offset al livello dBFS di riferimento
18   iFinalDbfs      = iDbfsRef1kHz + iFrequencyOffset
19
20 ; 6. Converti il dBFS finale in ampiezza lineare
21   iFinalAmp       = ampdbfs(iFinalDbfs)
22
23 xout iFinalAmp
24 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:

```
1 opcode PhonToSpl_i, i, ii
2   iphon, ifreq   xin
3
4 ; Interpolazione lineare dalle tabelle ISO
5   iaf           Interp   ifreq, giIsoFreqs, giAf
6   ilu           Interp   ifreq, giIsoFreqs, giLu
7   itf           Interp   ifreq, giIsoFreqs, giTf
8
9 ; Formula ISO 226:2003
10  iterm1         =       4.47e-3 * (pow(10, 0.025 * iphon) - 1.15)
11  iterm2_exp     =       (itf + ilu) / 10.0 - 9
12  iterm2         =       pow(0.4 * pow(10, iterm2_exp), iaf)
```

```

13     iaf_value      =      iterm1 + iterm2
14
15 if iaf_value <= 0 then
16     ispl          =      itf + (iphon / 40.0) * 20
17 else
18     ispl          =      (10.0 / iaf) * log10(iaf_value) - ilu + 94.0
19 endif
20
21 if abs(ifreq - 1000) < 0.1 then
22     ispl = iphon
23 endif
24
25 xout          ispl
26 endop

```

La formula si divide in due termini:

- **item1**: Rappresenta la componente lineare della loudness, dominante a livelli alti
- **item2**: Cattura la non-linearità vicino alla soglia uditiva

Il caso speciale if $iaf_value \leq 0$ 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:

```

1 if iAmpStart > iAmpEnd then
2     kf expseg 1, p3, 0.0001
3     kFinalAmp = (kf * (iAmpStart-iAmpEnd))+iAmpEnd
4 elseif iAmpStart < iAmpEnd then
5     kf expseg 0.0001, p3, 1
6     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:

```
1 ; Parametri di base per la spazializzazione
2 iwhichZero = abs(p6)      ; quale "zero" della funzione trigonometrica usare
3 iHR = max(1, abs(p7))    ; Harmonic Ratio - il numero di "spicchi" della
    ↪ circonferenza
4
5 ; Calcolo del periodo e della posizione iniziale
6 iPeriod = $M_PI * 2 / iHR
7 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:

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

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

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

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

La formula $\text{abs}(\sin(\text{krad} * i_{\text{HR}} / 2))$ genera curve polari modificate. Questa trasformazione:

- Prende il valore assoluto, creando lobi sempre positivi
- Moltiplica per $i_{\text{HR}} / 2$, 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:

```
1 def genera_e_plotta_polare_sine(self):
2     theta = np.linspace(0, 2 * np.pi, 500)
3     num_funzioni = 10
4     \section{Base delle funzioni sinusoidali}
5     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 involuppo sempre più complessi.

La conversione finale da coordinate polari a stereo avviene con:

```
1 ; Calcolo delle componenti Mid-Side
2 kMid = cos(krad)
3 kSide = sin(krad)
4
5 ; Applicazione dell'involuppo al segnale
6 aMid = kMid * asigEnv
7 aSide = kSide * asigEnv
8
9 ; Conversione a Left-Right con matrice di rotazione
10 aL = (aMid + aSide) / $SQRT2
11 aR = (aMid - aSide) / $SQRT2
```

2.2.3 Gestione degli Involuppi Multipli

Il sistema gestisce due livelli di involuppo che interagiscono moltiplicativamente:

```
1 ; Involuppo locale dell'evento (derivato dalle armoniche spaziali)
2 asigLocalEnv = asig * kEnv_local
3
4 ; Involuppo di sezione (se presente)
5 kEnv_section = 1
6 if i_ifn_section_env > 20 && i_section_duration > 0 then
7     k_time_absolute = times
8     k_time_since_section_start = k_time_absolute - i_section_start_time
9     kndx_section = limit(k_time_since_section_start / i_section_duration, 0, 1)
10    kEnv_section = tablei_kndx_section, i_ifn_section_env, 1
11 endif
12
```

```

13 ;_Combinazione_degli_inviluppi
14 asigEnvPre=_asigLocalEnv*_kEnv_section
15 asigEnv_dcblock_asigEnvPre

```

L'involuppo di sezione permette modulazioni globali su tutti gli eventi di una sezione, mentre l'involuppo 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, gestita dall'opcode GenPythagFreqs:

```

1 opcode GenPythagFreqs, i, iii
2   iFund, iNumIntervals, iNumOctaves, iTblNum xin
3   iTotallen = iNumIntervals * iNumOctaves
4   iFreqs[] init iTotallen
5
6   iOctave = 0
7   iBaseIndex = 0
8
9   while iOctave < iNumOctaves do
10     iFifth = 3/2
11     iFreqs[iBaseIndex] = iFund * (2^iOctave)
12
13 ; Genera la serie di quinte per questa ottava
14     indx = 1
15     iLastRatio = 1
16     while (indx < iNumIntervals) do
17       iRatio = iLastRatio * iFifth
18       ; Riduci all'ottava di riferimento
19       while (iRatio >= 2) do
20         iRatio = iRatio / 2
21       od
22       iFreqs[iBaseIndex+indx] = iFund * iRatio * (2^iOctave)
23       iLastRatio = iRatio
24       indx += 1
25     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:

```

1 ; Ordina le frequenze per questa ottava
2 indx = iBaseIndex

```

```

3 while (indx < (iBaseIndex + iNumIntervals - 1)) do
4   indx2 = indx + 1
5   while (indx2 < (iBaseIndex + iNumIntervals)) do
6     if (iFreqs[indx2] < iFreqs[indx]) then
7       iTemp = iFreqs[indx]
8       iFreqs[indx] = iFreqs[indx2]
9       iFreqs[indx2] = iTemp
10    endif
11    indx2 += 1
12  od
13  indx += 1
14 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:

```

1 opcode calcFrequenza, i, iii
2   i_Ottava, i_Registro, i_RitmoCorrente xin
3
4 ; Calculate octave register
5   i_Indice_Ottava = int(i_Ottava * $INTERVALLI)
6   ; Calculate interval offset within the octave
7   i_OffsetIntervallo = i_Indice_Ottava + int(((i_Registro * $INTERVALLI) /
      ↪ $REGISTRI))
8
9 ; Get the frequency from the table using the calculated offset
10  i_Freq table max(1, i_OffsetIntervallo + i_RitmoCorrente), gi_Intonazione
11  ifreq = min(i_Freq, sr/2-1)
12  xout ifreq
13 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] non solo definisce:

- Le durate primarie relative degli eventi ($\text{durataArmonica}/3$, $\text{durataArmonica}/5$, 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'involuppo 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 sonoro con battimenti e risonanze particolari.

La gerarchia Voce → eventoSonoro, supportata dal sistema di intonazione pitagorica, dalle tecniche di compensazione isofonica e dal sistema di armoniche spaziali, fornisce al compositore uno strumento di straordinaria flessibilità espressiva, capace di generare texture complesse da specifiche relativamente semplici.

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*

```

1 opcode NonlinearFunc, i, ippo
2   iX, iMode, iMinVal, iMaxVal xin
3
4 ; Valori di default per min/max se non specificati
5   iMinVal = (iMinVal == 0) ? 1 : iMinVal
6   iMaxVal = (iMaxVal == 0) ? 35 : iMaxVal
7
8 ; Assicuratevi che iX sia entro limiti sensati
9   iX = limit(iX, 1, 100)
10
11 iPI = 4 * taninv(1.0)
12   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

```
1 if iMode == 0 then
2     ; --- MODALITÀ 0: CONVERGENTE ---
3     iR = 2.8
4     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:

```
1 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

```
1 elseif iMode == 1 then
2     ; --- MODALITÀ 1: PERIODICA ---
3     iP1 = sin(iX * iPI/18)
4     iP2 = cos(iX * iPI/10)
5     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:

- $\sin(iX * \frac{\pi}{18})$: periodo di 36 unità
- $\cos(iX * \frac{\pi}{10})$: 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 $\text{abs}(iP1 * iP2) * 20 + 10$:

- 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

```

1 elseif iMode == 2 then
2   ; --- MODALITÀ 2: CAOTICA DETERMINISTICA ---
3   iR = 3.99
4   iNormX = (iX % 100) / 100
5   iNormX = limit(iNormX, 0.01, 0.99)
6   iLogistic = iR * iNormX * (1 - iNormX)
7   iNoise = random:i(-0.05, 0.05)
8   iLogistic = limit(iLogistic + iNoise, 0, 1)
9   iRange = iMaxVal - iMinVal + 1
10  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)

```

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

```

1      ; 2. Componente casuale (40%)
2      iRandom = random:i(0, 1)
3
4 ; 3. Combina le componenti
5      iMixRatio = 0.6
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

```

1      ; 4. Perturbazione periodica
2      iPerturbation = 0
3      if (iX % 7 == 0) then
4          iPerturbation = random:i(-0.3, 0.3)
5      endif
6
7 ; 5. Mappa al range finale
8      iRange = iMaxVal - iMinVal + 1
9      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 $\pm 30\%$ 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:

```

1 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 alitmicamente

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 involuppo utilizzate nella sintesi. La sua struttura gerarchica separa chiaramente gli involuppi per eventi singoli da quelli per sezioni intere.

3.1.1 Struttura delle Definizioni di Tabella

Ogni tabella è definita attraverso quattro parametri fondamentali:

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

Vediamo un esempio concreto:

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

Questa definizione genera in Csound:

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

La scelta della GEN routine 6 (segmenti cubici) invece della più comune GEN 7 (segmenti lineari) permette transizioni più morbide tra i punti di controllo, essenziale per inviluppi naturali.

3.1.2 *Inviluppi Evento vs Inviluppi Sezione*

Il sistema distingue due categorie di inviluppi con funzioni distinte:

Event Envelopes Applicati ai singoli eventi sonori:

```
1 event_envelopes:
2   impulsivo:
3     number: 5
4     size: 4096
5     gen_routine: 5
6     parameters: [0.001, 512, 1, 3584, 0.0001]
7
8   lento:
9     number: 6
10    size: 4096
11    gen_routine: 7
12    parameters: [0, 3072, 1, 1024, 0]
13
14  sostenuto:
15    number: 7
16    size: 4096
17    gen_routine: 7
18    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 crescendo gradualmente.

Section Envelopes Modulano interi gruppi di eventi:

```
1 section_envelopes:
2   crescendo_diminuendo:
3     number: 24
4     size: 4096
5     gen_routine: 7
6     parameters: [0, 2048, 1, 2048, 0]
7
8   impulso:
9     number: 25
10    size: 4096
11    gen_routine: 6
12    parameters: [1, 4096, 0.001]
```

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

```
1 if i_ifn_section_env > 20 && i_section_duration > 0 then
2     ; Applica involucro di sezione
3 endif
```

3.2 Sistema di Macro e Costanti Globali

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

```
1 #define FONDAMENTALE #32#
2 #define OTTAVE #10#
3 #define INTERVALLI #200#
4 #define REGISTRI #50#
```

3.2.1 Parametri dello Spazio Frequenziale

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
```

3.2.2 Integrazione con Python

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 involucri 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:

```
1 def load_all_compositions_from_yaml(file_path):  
2     """  
3     Carica una o più composizioni da un singolo file YAML.  
4     I documenti multipli devono essere separati da '---'.  
5     Restituisce una lista di strutture di composizione.  
6     """
```

```

7      print(f"Caricamento partiture dal file multi-documento: {file_path}")
8      composizioni = []
9      try:
10         with open(file_path, 'r') as f:
11 \section{safe_load_all restituisce un generatore, lo convertiamo in lista}
12             docs = list(yaml.safe_load_all(f))
13 \section{... (gestione errori) ...}
14             for i, composition in enumerate(docs):
15                 if composition is None: continue
16                 composizioni.append(composition)
17             return composizioni
18 \section{... (gestione eccezioni) ...}

```

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 involucri (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.

4.2.1 Struttura a Layer e `_process_layer`

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.

4.2.2 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`.

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

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

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

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

```
1 elif 'choices' in p_mask:
2     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.

4.2.3 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.

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

4.2.4 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 involucri.
2. **Eventi (i-statements):** Itera su ogni evento generato e scrive una riga di score (i "Voce" ...) con tutti i parametri calcolati.

```
1 \section{Frammento della riga di score generata}
2 score_lines += (f'i_Voce"\t{event_time:.4f}\t{p["durata_totale"]:.3f}\t'
3                 f'{p["ritmi_tab_num"]}\t{p["durata_armonica"]:.3f}\t\t{p["
4                 ↪ dynamic_index"]:.6f}\t'
5 \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**: Scomponi 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. Se l'orchestra Csound è il corpo esecutore, i metodi analizzati in questo capitolo rappresentano il cervello decisionale che lo pilota. 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.

```
1 SKIPPED_KEYS = {
2     'choices', 'weights', 'distribution', # Metadati per la generazione
3     'dynamic_index', 'dinamica',         # Logica di dinamica complessa
4     'nonlinear_mode', 'senso_movimento', # Parametri di controllo per Csound
5     'inviluppo_attacco', 'tipo_ritmi',    # Richiedono traduzione o
6     ↪ generazione complessa
7     'densita_cluster'
8 }
```

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

5.1.2 Le Modalità di Generazione: Un Toolkit Espressivo

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

```
1 for key, p_mask in mask.items():
2     if key in SKIPPED_KEYS: continue
3     \section{1. Distribuzione Normale (Gaussiana)}
4     if 'mean' in p_mask and 'std' in p_mask:
5         val = np.random.normal(loc=p_mask['mean'], scale=p_mask['std'])
6     \section{2. Distribuzione Uniforme (Range)}
7     elif 'range' in p_mask:
8         min_val, max_val = p_mask['range']
9         val = random.randint(min_val, max_val) if isinstance(min_val, int) else
10        ↪ random.uniform(min_val, max_val)
11 \section{3. Scelta Discreta Pesata}
12 elif 'choices' in p_mask:
13     val = random.choices(p_mask['choices'], weights=p_mask.get('weights'), k
14     ↪ =1)[0]
15 \section{4. Valore Fisso}
16 elif 'value' in p_mask:
17     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 : Utile quando tutti i valori in un intervallo sono ugualmente desiderabili. Un range: [1, 10] per il registro crea salti discreti, mentre un range: [1.0, 10.0] abilita micro-intervalli, mostrando come una scelta tecnica influenzi direttamente il risultato sonoro.
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.

```

1 if 'explicit_values' in rhythm_mask:
2 \section{Modalità 1: Controllo totale con una lista esplicita}
3     params['ritmi'] = rhythm_mask['explicit_values']
4 elif 'choices' in rhythm_mask:
5     choice = random.choices(rhythm_mask['choices'], ...)[0]
6     if isinstance(choice, list):
7 \section{Modalità 2: Scelta tra pattern pre-composti}
8         params['ritmi'] = choice
9     else:
10 \section{Modalità 3: Astrazione massima tramite categorie ('piccoli', 'medi'...)}
11     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 3.2 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 con una particolare attenzione alla robustezza e alla coerenza musicale.

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.

```

1 all_keys = set(start_mask.keys()) | set(end_mask.keys())
2 for key in all_keys:
3     s_mask = start_mask.get(key)
4     e_mask = end_mask.get(key)
5
6 if s_mask is None: s_mask = e_mask
7     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 crea 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 *buste 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 3.3 Un Caso di Studio: 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.

In conclusione, l'architettura di generazione parametrica di Gamma è stratificata e flessibile. Separa nettamente la specifica compositiva (l'intenzione nel YAML) dall'implementazione tecnica (la logica di generazione e interpolazione in Python). Questo permette al compositore di operare al livello di astrazione più consona, che sia definire un pattern nota per nota o semplicemente tracciare una *tendenza* evolutiva, lasciando al sistema il compito di tradurre questa intenzione in una texture sonora ricca e coerente.

6 SCOLPIRE IL TEMPO: MODELLI TEMPORALI E MICRO-RITMICA

Se i parametri definiscono il *cosa* di un evento sonoro, il controllo temporale ne definisce il *quando*. In un sistema generativo, questa dimensione assume un'importanza cruciale, poiché la distribuzione degli eventi nel tempo è uno dei principali veicoli dell'espressione musicale. Gamma affronta questa sfida con un approccio a due livelli: una macro-gestualità definita da modelli temporali astratti e una micro-ritmica che introduce variazioni organiche e *umanizza* la griglia computazionale.

6.1 La Macro-gestualità: TimeScheduler e i Modelli Temporali

Il posizionamento degli eventi (o meglio, dei cluster di eventi) all'interno di un layer non è lasciato al caso. 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.

```

1 def generate_onsets(self, model, duration, num_events):
2     \section{... (gestione casi base) ...}
3     base_progress = np.linspace(0, 1, num_events, endpoint=False)
4     final_progress = np.zeros_like(base_progress)

```



```

5     model_type = model.get('type', 'linear')
6 \section{... applicazione del modello ...}
7     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.

6.1.1 I Modelli Archetipici

1. 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.
2. Ritardando (type: ritardando): Simula un rallentamento progressivo.

```

1 elif model_type == 'ritardando':
2     shape = model.get('shape', 2.0)
3     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 più drammatico e pronunciato.

1. Accelerando (type: accelerando): Speculare al ritardando, crea un aumento di tensione.

```

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

```

La formula inverte la progressione, applica la potenza e la inverte nuovamente. Il risultato è una curva che parte lentamente e accelera in modo esponenziale, ideale per creare archi di tensione, approcci a un climax o gesti di *accumulazione* energetica.

1. Stocastico (type: stochastic): Per texture organiche e imprevedibili.

```

1 elif model_type == 'stochastic':
2     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 e imitare processi naturali o *pointillistici*.

6.1.2 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:

```
1 timing_model:
2   type: breakpoint
3   points:
4   \begin{itemize}
5     \item [0.0, 0.0]      # Inizio
6     \item [0.3, 0.7, 0.5] # Il 70% degli eventi avviene nel primo 30% del
7       ↪ tempo (curva concava, ease-out)
8     \item [1.0, 1.0, 3.0] # Il restante 30% degli eventi si distribuisce nel
7       ↪ 70% del tempo rimanente (curva convessa, ease-in)
8   \end{itemize}
```

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:

```
1 \section{... (all'interno del loop sui segmenti) ...}
2 \section{1. Normalizza il tempo all'interno del segmento (da 0 a 1)}
3 time_in_segment = (segment_times - t_start) / (t_end - t_start)
4 \section{2. Applica la funzione di shaping (curva)}
5 shaped_time = time_in_segment ** shape
6 \section{3. Interpola linearmente tra i VALORI usando il tempo curvato}
7 interpolated_values = v_start + (v_end - v_start) * shaped_time
8 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

YAML (YAML Ain't Markup Language) emerge in Gamma non solo come formato di configurazione, ma come vero e proprio linguaggio di partitura per la composizione algoritmica. La scelta di YAML rispetto ad altri formati riflette la necessità di bilanciare leggibilità umana con precisione computazionale, creando un ponte tra l'intuizione compositiva e l'esecuzione algoritmica.

7.1 Struttura Gerarchica

La struttura compositiva in Gamma segue una gerarchia rigorosa che rispecchia l'organizzazione tradizionale della musica occidentale, adattandola alle esigenze della generazione algoritmica.

7.1.1 La Gerarchia Fondamentale

Al livello più alto, una composizione è una lista di sezioni:

```
1 \begin{itemize}
2   \item nome_sezione: "Introduzione"
3 \end{itemize}
4   durata: 30
5   layers:
6 \begin{itemize}
7   \item nome_layer: "Texture_di_base"
8 \end{itemize}
9 \section{parametri del layer}
10 \begin{itemize}
11   \item nome_layer: "Eventi_puntuali"
12 \end{itemize}
13 \section{parametri del layer}
14 \begin{itemize}
15   \item nome_sezione: "Sviluppo"
16 \end{itemize}
17   durata: 60
18   layers:
19 \section{altri layers}
```

Questa struttura apparentemente semplice nasconde una ricchezza semantica considerevole. Ogni livello gerarchico porta con sé un dominio di parametri specifico e regole di ereditarietà implicite.

7.1.2 Parametri per Livello Gerarchico

Livello Sezione: I parametri a questo livello influenzano tutti i layer contenuti:

- **durata:** Definisce il contenitore temporale
- **ratio_temporale:** Permette dilatazioni o compressioni senza modificare i valori numerici
- **inviluppo_sezione:** Applica una modulazione globale d'ampiezza

Livello Layer: Qui si definisce l'identità del flusso sonoro:

- `num_attivazioni`: Controlla la densità eventi
- `timing_model`: Determina la distribuzione temporale
- `lifespan`: Definisce quando il layer è attivo
- Stati (unico/iniziale/finale): Contengono le maschere di tendenza

La separazione dei domini parametrici non è arbitraria. Riflette una comprensione che certi aspetti musicali (come la durata totale di una sezione) sono strutturali, mentre altri (come la distribuzione delle altezze) sono texturali.

7.1.3 Eredità e Override dei Valori

Il sistema implementa un modello di eredità implicita dove i valori di default si propagano attraverso la gerarchia:

```

1 \begin{itemize}
2   \item nome_sezione: "Sezione□con□defaults"
3 \end{itemize}
4   durata: 60
5 \section{ratio_temporale assume valore 1.0}
6 \section{involuppo_sezione assume 'continua'}
7   layers:
8 \begin{itemize}
9   \item nome_layer: "Layer□minimale"
10 \end{itemize}
11 \section{num_attivazioni assume 10}
12 \section{timing_model assume}
13   stato_unico:
14     ottava: {range: [4, 6]}
15 \section{tutti gli altri parametri assumono defaults}

```

Questa eredità permette specifiche concise quando i defaults sono appropriati, ma mantiene la possibilità di override granulare quando necessario. Il meccanismo di normalizzazione in Python garantisce che anche parametri specificati in forma abbreviata vengano espansi nella forma completa prima del processing.

7.2 Definizione delle Maschere

Le maschere di tendenza rappresentano il cuore semantico del sistema, trasformando il YAML da semplice formato di dati a linguaggio espressivo per la composizione.

7.2.1 Sintassi delle Modalità di Generazione

La sintassi per le maschere supporta quattro modalità principali, ciascuna con la propria semantica:

Range: Definisce un intervallo di valori equiprobabili:

```
1 ottava: {range: [3, 7]}
2 registro: {range: [1.0, 10.0]} # float permette microtonalità
```

Choices: Permette selezione da un insieme discreto:

```
1 dinamica: {choices: ['p', 'mf', 'f']}
2 \section{Con pesi per distribuzione non uniforme}
3 tipo_ritmi: {choices: ['piccoli', 'medi'], weights: [0.3, 0.7]}
```

Distribuzione Normale: Per concentrazione attorno a un centro:

```
1 durata_armonica: {mean: 2.0, std: 0.5}
```

Valore Fisso: Quando non si desidera variazione:

```
1 senso_movimento: {value: -1}
```

7.2.2 Normalizzazione Automatica

Il sistema permette sintassi abbreviate che vengono espanso automaticamente:

```
1 \section{Forma abbreviata}
2 dinamica: 'mf'
3 \section{Viene normalizzata in}
4 dinamica: {value: 'mf'}
```

Questa normalizzazione avviene nel metodo `_normalize_mask()` e permette di mantenere il YAML leggibile senza sacrificare la consistenza interna del sistema.

7.2.3 Parametri Interpolabili vs Fissi

Non tutti i parametri supportano l'interpolazione. La distinzione riflette la natura musicale dei parametri:

Interpolabili:

- Parametri numerici continui (ottava, registro, durata)
- Distribuzioni (mean, std di una normale)
- Pesi di scelte discrete (quando le scelte sono identiche)

Non Interpolabili:

- Stringhe che rappresentano categorie (`tipo_ritmi` quando usa categorie)
- Liste di valori (`explicit_values` per ritmi)
- Parametri strutturali (`timing_model`)

Questa distinzione è gestita automaticamente dal sistema di interpolazione, che applica strategie appropriate per ogni tipo.

7.3 Controlli Avanzati

Oltre ai parametri musicali di base, Gamma offre controlli avanzati che permettono di gestire aspetti sottili ma cruciali della generazione.

7.3.1 Lifespan e Ciclo di Vita dei Layer

Il parametro `lifespan` permette controllo fine su quando un layer è attivo:

```

1 layers:
2   \begin{itemize}
3     \item nome_layer: "Introduzione_□graduale"
4   \end{itemize}
5     lifespan: [0.0, 0.3] # Solo nel primo 30%
6
7   \begin{itemize}
8     \item nome_layer: "Corpo_□principale"
9   \end{itemize}
10     lifespan: [0.2, 0.9] # Dal 20% al 90%, sovrapposizione con intro
11
12   \begin{itemize}
13     \item nome_layer: "Coda"
14   \end{itemize}
15     lifespan: [0.8, 1.0] # Ultimo 20%, sovrapposizione con corpo

```

Questa specifica crea una forma ad arco con sovrapposizioni controllate, impossibile da ottenere con semplice sequenzialità.

7.3.2 Safety Buffer e Leeway

Due meccanismi complementari gestiscono i bordi temporali:

```

1 \begin{itemize}
2   \item nome_layer: "Eventi_□lunghi"
3 \end{itemize}
4   usa_safety_buffer: true # Default, previene sconfinamenti
5   leeway_fine_layer: 2.0 # Permette 2 secondi extra alla fine

```

Il safety buffer sottrae tempo dalla generazione per garantire che nessun evento ecceda i limiti. Il leeway aggiunge tempo extra alla fine per permettere code naturali. La combinazione permette controllo preciso del comportamento ai bordi mantenendo flessibilità espressiva.

7.3.3 *Modalità Solo e Veteran Mode*

Due modalità speciali facilitano il workflow compositivo:

```
1 \begin{itemize}
2   \item nome_layer: "Layer_in_sviluppo"
3 \end{itemize}
4 solo: true # Solo questo layer sarà renderizzato
5
6 \begin{itemize}
7   \item nome_layer: "Layer_completato"
8 \end{itemize}
9 veteranMode: true # Riuserà il WAV esistente se presente
```

La modalità solo isola layer specifici per testing e rifinitura. Il veteranMode ottimizza i tempi di rendering riutilizzando materiale già generato. Entrambe le modalità operano a livello di orchestrazione Python senza influenzare la generazione Csound.

7.3.4 *Implicazioni Compositive della Sintassi*

La sintassi YAML di Gamma non è neutra - incorpora assunzioni e affordance che guidano il processo compositivo. La struttura gerarchica suggerisce di pensare in termini di sezioni e layer. La sintassi delle maschere incoraggia il pensiero probabilistico. I controlli avanzati permettono raffinamenti che sarebbero complessi in notazione tradizionale.

Questa non è semplicemente una questione di convenienza. Il linguaggio che usiamo per descrivere la musica influenza profondamente come pensiamo alla musica stessa. YAML in Gamma diventa così non solo un formato di dati, ma un medium compositivo che shapes il pensiero musicale verso paradigmi stocastici e stratificati, mantenendo al contempo una connessione con i concetti tradizionali di struttura e forma musicale.

8 CONCLUSIONE: UN ECOSISTEMA PER LA COMPOSIZIONE

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 princi-

pale 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 layer, trasformando la composizione da un insieme di processi paralleli a un vero e proprio ecosistema 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.

In definitiva, Gamma si è dimostrato una piattaforma robusta e flessibile, un fondamento tecnico e concettuale solido. L'esperienza acquisita nella sua progettazione e nel suo utilizzo non è solo un traguardo, ma il trampolino di lancio per esplorare la prossima frontiera della musica generativa: non più solo creare suono secondo regole, ma progettare sistemi che scoprono le proprie regole attraverso l'esplorazione e l'interazione.