INDIAN INSTITUTE OF TECHNOLOGY, BOMBAY

Chord Engine using linear Algebra

A Vector Space Approach to Music Generation Using 12-TET

Submitted By:

Rishi Raj Vishwakarma	Roll No: 25M2161
Adithi Roy Chowdhury	Roll No: 25M2167
Samridh Agarwal	Roll No: 25M2156
Deepak Mewada	Roll No: 25M2158

Under Course:

Applied Linear Algebra (EE 635)
Course Instructor: Prof. Debasattam Pal

Academic Year: 2024–2025

Date of Submission: November 27, 2025

Contents

Abstract

Music theory is traditionally expressed through symbolic notation and auditory interpretation. This project presents a mathematical framework that models musical notes, chords, and progressions using concepts from Linear Algebra. By representing the twelve pitch classes of Western music as orthonormal basis vectors in $\mathbb{R}^{12}$, we construct chords using linear combinations and define transposition through cyclic shift matrices. The project further introduces chord operators that generate major, minor, and extended harmonies as matrix transformations acting on single-note vectors. Frequency realization is achieved through octave transformation matrices that map abstract pitch-class vectors to audio signals. Temporal information is incorporated using beat–duration scaling, allowing chord progressions to be synthesized at any tempo. Finally, we develop a sound synthesis pipeline that employs overtone-based additive synthesis to approximate piano timbre and generate audible chord progressions. The resulting system demonstrates that musical composition and playback can be expressed as linear transformations, showing the power and applicability of Linear Algebra in digital music generation.

Chapter 1

Introduction

1.1 Motivation

The idea for this project emerged while studying the topic of **solution parametrisation of linear systems** in Linear Algebra, where multiple solutions can be represented as vectors belonging to the same subspace. This inspired the question of whether musical elements, particularly notes and chords, could also be represented in a vector space and manipulated using linear transformations.

Initially, a two–dimensional encoding was attempted, where each note was represented as a vector in $\mathbb{R}^2$ of the form:

$$[\log_2(\text{frequency}), \text{octave}]$$

However, this model proved inadequate for harmonic construction, since notes defined by frequency and octave do not form a linear space under addition, and the representation failed to capture *pitch-class equivalence* and transposition correctly.

Later topics covered in class, such as **Basis, Kernel, and Linear Transformations**, motivated a shift towards a more abstract representation. This led to the adoption of a model in $\mathbb{R}^{12}$, where each of the twelve pitch classes in Western music corresponds to a basis vector. In this higher-dimensional space, chords can be constructed using linear combinations of basis vectors, and musical operations such as transposition can be expressed as linear transformations using shift matrices. Thus, the key motivation evolved from realizing that musical harmony can be generated and manipulated entirely through the tools of Linear Algebra.

1.2 Key Idea

The central idea of this project is to model musical notes, chords, and harmonic motion using the algebraic structure of vector spaces and linear transformations. Instead of interpreting music through symbolic notation or frequency ratios, our approach treats musical objects as vectors in $\mathbb{R}^{12}$, where each dimension corresponds to one of the twelve pitch classes of Western equal temperament:

$$\{\mathrm{C}, \mathrm{C}^\sharp, \mathrm{D}, \mathrm{D}^\sharp, \mathrm{E}, \mathrm{F}, \mathrm{F}^\sharp, \mathrm{G}, \mathrm{G}^\sharp, \mathrm{A}, \mathrm{A}^\sharp, \mathrm{B}\}$$

Pitch Classes as Basis Vectors

Each pitch class is represented as a standard basis vector e_i in $\mathbb{R}^{12}$:

$$e_0 = (1, 0, 0, \ldots, 0)^T, \quad e_1 = (0, 1, 0, \ldots, 0)^T, \quad \ldots, \quad e_{11}$$

This representation allows musical objects to be stored using purely algebraic constructs without involving frequency or octave at this stage.

Chords as Linear Combinations

A chord contains multiple pitch classes. In this model, a chord is simply a linear combination of the corresponding basis vectors. For instance, a major triad consisting of the root, major third, and perfect fifth is represented as:

$$\text{Major Triad} = e_r + e_{(r+4) \bmod 12} + e_{(r+7) \bmod 12}$$

where r is the root pitch class index. This model naturally supports chord construction by addition of basis vectors.

Transposition via Linear Operators

Musical transposition is expressed as a linear transformation using a cyclic *shift matrix* $S \in \mathbb{R}^{12\times 12}$. The matrix shifts one pitch class to the next:

$$Se_k = e_{(k+1) \bmod 12}$$

Thus, transposing a note or chord by m semitones is equivalent to applying S^m. For example:

$$S^4(e_r) = e_{(r+4) \bmod 12}$$

which retrieves the major third above r. Using this matrix, entire chord families can be expressed as *operators*, such as the major chord operator:

$$M_{\text{maj}} = I + S^4 + S^7$$

Frequency Synthesis via Transformation

Though chord operations occur in pitch-class space, audio must be generated in the domain of frequencies. A separate *octave transformation matrix* maps a pitch-class vector into a vector of real frequencies in Hz. This final step allows the abstract chord vectors to be converted into numerical signals suitable for sound synthesis in Python.

Overall, the key idea is to separate musical logic from acoustic realization: *harmony is computed in* $\mathbb{R}^{12}$ *using linear algebra, and audio synthesis is performed after mathematical transformations.* This separation enables chord generation, transposition, and playback to be performed entirely through matrix operations.

1.3 Mathematical Modeling

The mathematical framework of this project is built on three key tools from Linear Algebra: **(i) vector space representation of pitch classes**, **(ii) matrix operators for chord construction and transposition**, and **(iii) a frequency transformation for audio synthesis**. All musical computation is performed in $\mathbb{R}^{12}$, while sound is generated only after the mathematical transformations.

1.3.1 Pitch-Class Vector Space

We consider the twelve pitch classes of Western music as orthonormal basis vectors in the vector space $\mathbb{R}^{12}$. Each pitch class corresponds to exactly one standard basis vector:

$$\mathbf{e}_0 = (1,0,0,\dots,0)^T,\ \mathbf{e}_1 = (0,1,0,\dots,0)^T,\ \dots,\ \mathbf{e}_{11}$$

Any individual note in a scale or chord is a vector $\mathbf{e}_i$, and any group of notes is represented as a linear combination of these basis vectors. Therefore, a chord is expressed mathematically as:

$$\mathbf{c} = \sum_{i \in K} \mathbf{e}_i$$

where $K \subseteq \{0, 1, \ldots, 11\}$ is the set of pitch-class indices in the chord.

1.3.2 Chord Operators

Chord construction is expressed using matrix operators. Let $S \in \mathbb{R}^{12\times 12}$ denote the *shift matrix*, which cyclically maps a pitch class to the next one:

$$S\mathbf{e}_k = \mathbf{e}_{(k+1) \bmod 12}$$

For any $m \in \mathbb{Z}$, transposing by m semitones is achieved by S^m. This allows chords to be defined as *linear transformations*. For example, the major triad operator is:

$$M_{\text{maj}} = I + S^4 + S^7$$

Acting on a root note $\mathbf{e}_r$, the chord vector becomes:

$$\mathbf{c}_{\text{maj}} = M_{\text{maj}}\mathbf{e}_r$$

Similarly, the minor triad is:

$$M_{\text{min}} = I + S^3 + S^7$$

This formalism extends naturally to seventh chords, suspended chords, and jazz extensions. For example:

$$M_{\text{maj7}} = I + S^4 + S^7 + S^{11}$$

1.3.3 Frequency Transformation

To convert abstract chord vectors into audible tones, we define a frequency mapping using an octave transformation matrix $T_{\text{oct}} \in \mathbb{R}^{12\times 12}$. This matrix maps each basis vector $\mathbf{e}_i$ to the frequency (in Hz) of the corresponding pitch class in a specific octave:

$$T_{\text{oct}}\mathbf{e}_i = f_0 \cdot 2^{\frac{i+n\cdot 12}{12}}$$

where f_0 is a reference frequency (commonly A4 = 440 Hz), and n denotes the octave number. Applying this matrix to a chord vector produces a frequency vector:

$$\mathbf{f} = T_{\text{oct}}\mathbf{c}$$

These frequencies are then passed to a sound synthesis function to generate audio.

In summary, the mathematical model cleanly separates musical structure from sound synthesis: *all harmony is computed through linear transformations in $\mathbb{R}^{12}$, and only afterward mapped into the frequency domain for playback.*

1.4 Implementation Overview

The mathematical model was implemented entirely in Python using NumPy for matrix operations and `sounddevice` for audio playback. The system follows a modular workflow:

1. **Pitch Representation:** Notes are stored as one-hot vectors in $\mathbb{R}^{12}$ using a dictionary mapping each pitch class to its basis vector.
2. **Chord Generation:** Given a root note, chord vectors are computed by multiplying the root vector with chord operators such as $M_{\text{maj}}, M_{\text{min}}, M_{\text{maj7}}$, etc., constructed from powers of the shift matrix S.
3. **Frequency Mapping:** The chord vector is transformed to real frequencies by multiplying with the octave matrix T_{oct}, which assigns frequencies to each pitch class for a given octave.
4. **Audio Synthesis:** The resulting frequency set is synthesized using additive synthesis with harmonic overtones to approximate a piano timbre. Durations are computed using beat scaling:

$$\text{time} = \frac{60}{\text{BPM}} \times \text{beats}$$

5. **Playback and Export:** Chord progressions are played using `sounddevice` and can be saved as WAV files using `soundfile`.

This implementation ensures that every chord and progression heard is generated strictly through linear algebraic transformations rather than stored musical data.

1.5 Outcome

The completed system demonstrates that musical harmony and sound generation can be achieved purely through linear algebraic operations. Every chord and progression produced in this project is constructed using matrix transformations in $\mathbb{R}^{12}$ rather than through stored frequencies or symbolic notation. The key outcomes are summarized below:

- Musical notes are represented as basis vectors, and chords are generated using linear combinations and operator matrices.
- Transposition, chord quality (major, minor, seventh, suspended), and harmonic extensions are performed using powers of a shift matrix.
- Octave-specific frequencies are obtained only after the algebraic computation, using a frequency transformation matrix.
- Chord progression playback is achieved through beat-based duration scaling and additive synthesis with piano-like overtones.
- The system can generate, play, transpose, and save any chord progression using the same mathematical framework, without manually storing musical structures.

Overall, the project validates that abstract concepts such as basis vectors, linear operators, and matrix transformations can be directly applied to a real-world domain like music, enabling automated audio synthesis through Linear Algebra.

Chapter 2

Mathematical Representation of Chords

2.1 Pitch Classes as a Vector Space

2.1.1 The Vector Space $\mathbb{R}^{12}$

We model pitch classes using the real vector space

$$\mathbb{R}^{12} = \{(x_1, x_2, \ldots, x_{12}) \mid x_i \in \mathbb{R}\}.$$

An element $\mathbf{x} \in \mathbb{R}^{12}$ is called a *12-dimensional vector* and is written as

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{12} \end{bmatrix}, \quad x_i \in \mathbb{R}.$$

Vector addition and scalar multiplication are defined component-wise:

$$\begin{aligned} \mathbf{x} + \mathbf{y} &= (x_1 + y_1,\ x_2 + y_2,\ \ldots,\ x_{12} + y_{12}), \\ \alpha\mathbf{x} &= (\alpha x_1,\ \alpha x_2,\ \ldots,\ \alpha x_{12}), \end{aligned}$$

for any $\mathbf{x}, \mathbf{y} \in \mathbb{R}^{12}$ and $\alpha \in \mathbb{R}$.

Vector Space Properties

We briefly justify that $\mathbb{R}^{12}$ with these operations forms a vector space over $\mathbb{R}$.

- **Closure:** For any $\mathbf{x}, \mathbf{y} \in \mathbb{R}^{12}$, their sum $\mathbf{x} + \mathbf{y}$ is still in $\mathbb{R}^{12}$ since each $x_i + y_i \in \mathbb{R}$. Similarly, for any scalar $\alpha \in \mathbb{R}$, $\alpha\mathbf{x} \in \mathbb{R}^{12}$ because each $\alpha x_i \in \mathbb{R}$.
- **Associativity and Commutativity of Addition:** Component-wise addition inherits associativity and commutativity from $\mathbb{R}$:

$$(\mathbf{x} + \mathbf{y}) + \mathbf{z} = \mathbf{x} + (\mathbf{y} + \mathbf{z}), \quad \mathbf{x} + \mathbf{y} = \mathbf{y} + \mathbf{x}.$$

- **Additive Identity:** The zero vector
$$\mathbf{0} = (0, 0, \ldots, 0)$$
satisfies $\mathbf{x} + \mathbf{0} = \mathbf{x}$ for all $\mathbf{x} \in \mathbb{R}^{12}$.
- **Additive Inverse:** For each $\mathbf{x} = (x_1, \ldots, x_{12})$, the vector $-\mathbf{x} = (-x_1, \ldots, -x_{12})$ satisfies
$$\mathbf{x} + (-\mathbf{x}) = \mathbf{0}.$$
- **Distributive and Scalar Associative Laws:** For any $\alpha, \beta \in \mathbb{R}$ and $\mathbf{x}, \mathbf{y} \in \mathbb{R}^{12}$,
$$\begin{aligned} \alpha(\mathbf{x} + \mathbf{y}) &= \alpha\mathbf{x} + \alpha\mathbf{y}, \\ (\alpha + \beta)\mathbf{x} &= \alpha\mathbf{x} + \beta\mathbf{x}, \\ (\alpha\beta)\mathbf{x} &= \alpha(\beta\mathbf{x}), \\ 1 \cdot \mathbf{x} &= \mathbf{x}. \end{aligned}$$

All of these follow directly from the distributive and associative properties of real numbers.

Since all vector space axioms are satisfied, $\mathbb{R}^{12}$ is a vector space over $\mathbb{R}$.

2.1.2 Standard Basis of $\mathbb{R}^{12}$

We now define the standard basis vectors of $\mathbb{R}^{12}$:

$$\mathbf{e}_1 = (1, 0, 0, \ldots, 0), \quad \mathbf{e}_2 = (0, 1, 0, \ldots, 0), \quad \ldots, \quad \mathbf{e}_{12} = (0, 0, \ldots, 0, 1).$$

Each $\mathbf{e}_i$ has a 1 in the i-th position and 0 in all other positions.

Spanning Property

Any vector $\mathbf{x} = (x_1, \ldots, x_{12}) \in \mathbb{R}^{12}$ can be written as a linear combination of these vectors:

$$\mathbf{x} = x_1\mathbf{e}_1 + x_2\mathbf{e}_2 + \cdots + x_{12}\mathbf{e}_{12}.$$

Therefore, the set $\{\mathbf{e}_1, \ldots, \mathbf{e}_{12}\}$ spans $\mathbb{R}^{12}$.

Linear Independence

Suppose we have a linear combination that equals the zero vector:

$$\alpha_1\mathbf{e}_1 + \alpha_2\mathbf{e}_2 + \cdots + \alpha_{12}\mathbf{e}_{12} = \mathbf{0}.$$

By expanding component-wise, this equation becomes:

$$(\alpha_1, \alpha_2, \ldots, \alpha_{12}) = (0, 0, \ldots, 0),$$

so each coefficient must be zero:

$$\alpha_1 = \alpha_2 = \cdots = \alpha_{12} = 0.$$

Hence, the only linear combination of $\{\mathbf{e}_1, \ldots, \mathbf{e}_{12}\}$ that gives $\mathbf{0}$ is the trivial one. Therefore, the set is linearly independent.

Conclusion: Basis of $\mathbb{R}^{12}$

Since the set $\{\mathbf{e}_1, \ldots, \mathbf{e}_{12}\}$ both spans $\mathbb{R}^{12}$ and is linearly independent, it forms a basis of $\mathbb{R}^{12}$.

2.1.3 Pitch Classes as Basis Vectors

In this project, we associate each of the twelve pitch classes with one of these basis vectors. For example, one possible mapping is:

$$\mathrm{C} \leftrightarrow \mathbf{e}_1,\ \mathrm{C}^\sharp \leftrightarrow \mathbf{e}_2,\ \ldots,\ \mathrm{B} \leftrightarrow \mathbf{e}_{12}.$$

A single pitch class is therefore represented as a basis vector in $\mathbb{R}^{12}$, and any collection of notes (such as a chord) is represented as a vector obtained by adding the corresponding basis vectors. This identification is the starting point for representing harmony using linear algebra.

2.1.4 Pitch-Class Encoding in $\mathbb{R}^{12}$

We assign each pitch class a unique basis vector in $\mathbb{R}^{12}$. The following mapping is used throughout this project, where each note corresponds to a one-hot vector:

$$\begin{aligned}
\mathrm{C} &\leftrightarrow (1,0,0,0,0,0,0,0,0,0,0,0)\\
\mathrm{C}^\sharp/\mathrm{D}^\flat &\leftrightarrow (0,1,0,0,0,0,0,0,0,0,0,0)\\
\mathrm{D} &\leftrightarrow (0,0,1,0,0,0,0,0,0,0,0,0)\\
\mathrm{D}^\sharp/\mathrm{E}^\flat &\leftrightarrow (0,0,0,1,0,0,0,0,0,0,0,0)\\
\mathrm{E} &\leftrightarrow (0,0,0,0,1,0,0,0,0,0,0,0)\\
\mathrm{F} &\leftrightarrow (0,0,0,0,0,1,0,0,0,0,0,0)\\
\mathrm{F}^\sharp/\mathrm{G}^\flat &\leftrightarrow (0,0,0,0,0,0,1,0,0,0,0,0)\\
\mathrm{G} &\leftrightarrow (0,0,0,0,0,0,0,1,0,0,0,0)\\
\mathrm{G}^\sharp/\mathrm{A}^\flat &\leftrightarrow (0,0,0,0,0,0,0,0,1,0,0,0)\\
\mathrm{A} &\leftrightarrow (0,0,0,0,0,0,0,0,0,1,0,0)\\
\mathrm{A}^\sharp/\mathrm{B}^\flat &\leftrightarrow (0,0,0,0,0,0,0,0,0,0,1,0)\\
\mathrm{B} &\leftrightarrow (0,0,0,0,0,0,0,0,0,0,0,1)
\end{aligned}$$

Each note is represented by a unique canonical basis vector. Any musical structure containing one or more notes (such as scales or chords) can therefore be represented as a vector obtained through addition of these pitch-class vectors.

2.2 Notes and Scales as Subspaces

In the vector space $\mathbb{R}^{12}$, a single musical note is represented as a canonical basis vector. When multiple notes are played together or belong to a common tonal system, they generate a subspace spanned by the corresponding basis vectors. Thus, both *notes* and *scales* can be interpreted as vector subspaces of $\mathbb{R}^{12}$.

2.2.1 Single Notes as One-Dimensional Subspaces

A single note, such as C, is represented by a basis vector $\mathbf{e}_1$. All scalar multiples of this vector form a one-dimensional subspace:

$$\operatorname{span}\{\mathbf{e}_1\} = \{\alpha\mathbf{e}_1 \mid \alpha \in \mathbb{R}\}.$$

Likewise, each pitch class forms a distinct one-dimensional subspace of $\mathbb{R}^{12}$.

2.2.2 Scales as Higher-Dimensional Subspaces

A musical scale consists of a fixed set of pitch classes. For example, the C major scale contains seven pitch classes:

$$\{\mathrm{C}, \mathrm{D}, \mathrm{E}, \mathrm{F}, \mathrm{G}, \mathrm{A}, \mathrm{B}\}.$$

In $\mathbb{R}^{12}$, this corresponds to the subspace spanned by the seven associated basis vectors:

$$\mathcal{S}_{\mathrm{Cmaj}} = \mathrm{span}\{\mathbf{e}_1, \mathbf{e}_3, \mathbf{e}_5, \mathbf{e}_6, \mathbf{e}_8, \mathbf{e}_{10}, \mathbf{e}_{12}\}.$$

Any melody or chord derived from the C major scale must lie within this subspace, as linear combinations of these seven vectors produce all valid pitch-class combinations in the scale.

2.2.3 Closure Under Addition

Let $\mathbf{x}$ and $\mathbf{y}$ be two pitch-class vectors belonging to the same scale subspace. Then any linear combination $\alpha\mathbf{x} + \beta\mathbf{y}$, where $\alpha, \beta \in \mathbb{R}$, remains within the span of the same basis vectors. Hence, scale vectors satisfy the requirements of a subspace:

$$\alpha\mathbf{x} + \beta\mathbf{y} \in \mathcal{S}_{\mathrm{scale}}.$$

2.2.4 Chord Membership in Scales

A chord contains notes belonging to a particular scale. For example, the C major triad,

$$\mathrm{Cmaj} = \{\mathrm{C}, \mathrm{E}, \mathrm{G}\},$$

is represented as:

$$\mathbf{c}_{\mathrm{Cmaj}} = \mathbf{e}_1 + \mathbf{e}_5 + \mathbf{e}_8,$$

which clearly lies in $\mathcal{S}_{\mathrm{Cmaj}}$. Thus, every chord vector belonging to a scale exists within the subspace spanned by that scale.

Therefore, musical notes correspond to basis vectors, and scales correspond to vector subspaces spanned by selected pitch classes in $\mathbb{R}^{12}$.

2.2.5 Chords as Direct Sums of Note Subspaces

In the vector space $\mathbb{R}^{12}$, each pitch class is represented by a canonical basis vector $\mathbf{e}_i$, where $i \in \{1, 2, \ldots, 12\}$. The vector space generated by a single note is therefore the one-dimensional subspace

$$V_i = \mathrm{span}\{\mathbf{e}_i\} \subset \mathbb{R}^{12}.$$

Definition 2.2.1 (Chord Subspace) *Let a chord contain k distinct pitch classes indexed by $\{i_1, i_2, \ldots, i_k\}$. The* chord subspace *is defined as*

$$V_{chord} = span\{\mathbf{e}_{i_1}, \mathbf{e}_{i_2}, \ldots, \mathbf{e}_{i_k}\}.$$

Theorem 2.2.1 *The subspace generated by a chord is the direct sum of the one-dimensional subspaces corresponding to its constituent pitch classes:*

$$V_{chord} = V_{i_1} \oplus V_{i_2} \oplus \cdots \oplus V_{i_k}.$$

Proof 2.2.1 *Since $\mathbf{e}_1, \mathbf{e}_2, \ldots, \mathbf{e}_{12}$ form the standard basis of $\mathbb{R}^{12}$, any subset of these vectors is linearly independent. In particular, for distinct indices $i_1, i_2, \ldots, i_k$, the vectors $\mathbf{e}_{i_1}, \mathbf{e}_{i_2}, \ldots, \mathbf{e}_{i_k}$ are linearly independent.*

Each $V_{i_j} = span\{\mathbf{e}_{i_j}\}$ is one-dimensional, and for $p \neq q$ we have

$$V_{i_p} \cap V_{i_q} = \{\mathbf{0}\},$$

since no nonzero scalar multiple of $\mathbf{e}_{i_p}$ can equal a scalar multiple of $\mathbf{e}_{i_q}$ when $i_p \neq i_q$.

To show the direct sum, we must verify that:

$$V_{chord} = V_{i_1} + V_{i_2} + \cdots + V_{i_k} \quad \textit{and} \quad V_{i_p} \cap \left(\sum_{q \neq p} V_{i_q} \right) = \{\mathbf{0}\}.$$

The first equality holds by definition of V_{chord}. The intersection property follows from the linear independence of the basis vectors. Since the sum of linearly independent one-dimensional subspaces with trivial intersections is a direct sum, we conclude that:

$$V_{chord} = V_{i_1} \oplus V_{i_2} \oplus \cdots \oplus V_{i_k}.$$

Thus, a chord subspace is the direct sum of the note subspaces it contains.

Example. The C major triad contains the pitch classes C, E and G. In $\mathbb{R}^{12}$ this corresponds to

$$V_{\text{Cmaj}} = \text{span}\{\mathbf{e}_1, \mathbf{e}_5, \mathbf{e}_8\} = \text{span}\{\mathbf{e}_1\} \oplus \text{span}\{\mathbf{e}_5\} \oplus \text{span}\{\mathbf{e}_8\}.$$

Hence, the chord vector

$$\mathbf{c}_{\text{Cmaj}} = \mathbf{e}_1 + \mathbf{e}_5 + \mathbf{e}_8$$

is an element of the direct sum that represents the chord in $\mathbb{R}^{12}$.

2.3 Shift Matrices for Transposition

In music, *transposition* means raising or lowering every note by a fixed number of semitones. In our vector model, this is done using a **shift matrix**, which rotates the pitch-class vector so that each note moves to the next one, just like shifting keys on a keyboard.

2.3.1 Shift Matrix Representation

We work in $\mathbb{R}^{12}$ with basis vectors

$$\mathbf{e}_1, \mathbf{e}_2, \ldots, \mathbf{e}_{12},$$

corresponding to the 12 pitch classes. The shift matrix S is the 12×12 matrix that maps each basis vector to the next one:

$$S\mathbf{e}_1 = \mathbf{e}_2, \quad S\mathbf{e}_2 = \mathbf{e}_3, \quad \ldots, \quad S\mathbf{e}_{12} = \mathbf{e}_1.$$

In matrix form, S is a cyclic permutation matrix with ones on the superdiagonal and a one in the bottom-left corner:

$$S = \begin{bmatrix} 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \\ 1 & 0 & 0 & \cdots & 0 \end{bmatrix}.$$

2.3.2 Effect on Notes

If the note C is represented by

$$\mathrm{C} = (1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)^T,$$

then multiplying by S gives:

$$S \cdot \mathrm{C} = (0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)^T,$$

which corresponds to C♯. Multiplying again moves to D, and so on. Shifting by n semitones is achieved by S^n.

2.3.3 Effect on Chords

A chord is represented as a sum of note vectors. For example, the C major chord

$$\mathrm{Cmaj} = (1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0)^T$$

contains the pitch classes C, E, and G. Applying the shift matrix gives

$$S \cdot \mathrm{Cmaj} = (0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0)^T,$$

which corresponds to the C♯ major chord (C♯–F–G♯). Thus, the same matrix transposes an entire chord with a single multiplication.

2.3.4 Practical Significance

- A single matrix can transpose any note, chord, or scale.
- All keys and chord positions are obtained by powers of the same operator S.
- Harmonic motion in music becomes a linear transformation in $\mathbb{R}^{12}$.

2.4 Linear Operators for Chords

Since each pitch class is represented by a basis vector in $\mathbb{R}^{12}$, a chord can be constructed by applying a matrix operator to a single note vector. In this way, chord formation becomes a linear transformation instead of a lookup or a stored rule.

2.4.1 Chord Operators

Let S denote the 12×12 shift matrix which raises a pitch by one semitone. Using S and its powers, we define chord operators by summing shifted versions of the identity matrix I.

Major Triad Operator. A major triad consists of the root, a major third (+4 semitones), and a perfect fifth (+7 semitones). This is represented by:

$$M_{\mathrm{maj}} = I + S^4 + S^7.$$

Minor Triad Operator. A minor triad has a minor third (+3 semitones) instead of +4:

$$M_{\mathrm{min}} = I + S^3 + S^7.$$

2.4.2 Generating Chords from Notes

If $\mathbf{e}_r$ denotes the root pitch class, then applying a chord operator yields the chord vector:

$$\mathbf{c}_{\text{maj}} = M_{\text{maj}}\mathbf{e}_r, \qquad \mathbf{c}_{\text{min}} = M_{\text{min}}\mathbf{e}_r.$$

Example. Let C be encoded as $\mathbf{e}_1$. Then the C major chord is computed as:

$$\mathbf{c}_{\text{Cmaj}} = M_{\text{maj}}\mathbf{e}_1 = \mathbf{e}_1 + \mathbf{e}_5 + \mathbf{e}_8,$$

representing the pitch classes C–E–G.

2.4.3 Extended Chords

The same construction naturally extends to seventh chords by adding their respective intervals:

$$M_{\text{maj7}} = I + S^4 + S^7 + S^{11}, \qquad M_{\text{min7}} = I + S^3 + S^7 + S^{10}.$$

These operators require no stored data: all chords are generated directly via matrix algebra.

2.4.4 Significance

- Any chord quality is encoded as a matrix, independent of the note played.
- All transpositions arise automatically using the same shift matrix.
- Chords become *linear transformations* acting on note vectors, not collections of stored notes.

Thus, harmony is modeled entirely through matrix operators, allowing music to be synthesized through linear algebra.

2.4.5 General Linear Operator for Chords

Let S denote the 12×12 shift matrix, and let $\mathbf{e}_r$ be the basis vector corresponding to the root pitch class r. If a chord quality is defined by a set of semitone intervals

$$K = \{k_1, k_2, \ldots, k_m\},$$

then the corresponding chord operator is defined as:

$$M_K = \sum_{k \in K} S^k,$$

where S^k denotes k successive semitone shifts. Applying this operator to a root note vector $\mathbf{e}_r$ generates the chord:

$$\boxed{\mathbf{c} = M_K\,\mathbf{e}_r}.$$

Thus, *any chord at any root* can be constructed using a matrix operator acting on a single note vector.

2.4.6 Chord Operator List

Triads

$$\begin{aligned}
M_{\text{maj}} &= I + S^4 + S^7 \\
M_{\text{min}} &= I + S^3 + S^7 \\
M_{\text{dim}} &= I + S^3 + S^6 \\
M_{\text{aug}} &= I + S^4 + S^8 \\
M_{\text{sus2}} &= I + S^2 + S^7 \\
M_{\text{sus4}} &= I + S^5 + S^7
\end{aligned}$$

Seventh Chords

$$\begin{aligned}
M_{\text{maj7}} &= I + S^4 + S^7 + S^{11} \\
M_{\text{min7}} &= I + S^3 + S^7 + S^{10} \\
M_{\text{dom7}} &= I + S^4 + S^7 + S^{10} \\
M_{\text{half-dim7}} &= I + S^3 + S^6 + S^{10} \\
M_{\text{dim7}} &= I + S^3 + S^6 + S^9
\end{aligned}$$

Sixth and Ninth Chords

$$\begin{aligned}
M_{\text{maj6}} &= I + S^4 + S^7 + S^9 \\
M_{\text{min6}} &= I + S^3 + S^7 + S^9 \\
M_{\text{maj9}} &= I + S^4 + S^7 + S^{11} + S^{14} \\
M_{\text{dom9}} &= I + S^4 + S^7 + S^{10} + S^{14}
\end{aligned}$$

Since pitch classes are cyclic, powers of S obey:

$$S^{n+12} = S^n,$$

so exponents such as S^{14} automatically wrap modulo 12.

All chord operators can be written concisely as:

$$\boxed{M_{\text{chord}} = \sum_{k \in K_{\text{chord}}} S^k}$$

where K_{chord} is the set of semitone intervals defining the chord quality. This representation avoids storing musical notes and generates all chords algebraically.

2.5 Frequency Transformation for Sound Synthesis

Chord vectors in $\mathbb{R}^{12}$ represent only pitch classes (note identities) and do not contain information about octave or physical frequency. To convert these abstract vectors into real sounds, we apply a linear transformation that maps each pitch class to its actual frequency in Hertz.

2.5.1 Octave Transformation Matrix

Let $T_{\text{oct}} \in \mathbb{R}^{12\times 12}$ be a diagonal matrix that assigns a frequency to each pitch class in a specific octave. If f_i denotes the frequency of the ith pitch class in that octave, then

$$T_{\text{oct}} = \text{diag}(f_1, f_2, \ldots, f_{12}).$$

Multiplying a pitch-class vector $\mathbf{x}$ by T_{oct} yields:

$$\mathbf{f} = T_{\text{oct}}\,\mathbf{x},$$

where $\mathbf{f}$ contains only the frequencies corresponding to active notes.

2.5.2 Frequency Formula

Each frequency f_i is computed using the equal temperament relation:

$$f_i = f_0\,2^{\frac{i-1}{12}+n},$$

where:

- f_0 is the reference frequency (A4 = 440 Hz),
- $i \in \{1, \ldots, 12\}$ indexes the pitch class,
- n denotes the octave number.

2.5.3 Mapping Chords to Sound

If $\mathbf{c}$ is a chord vector obtained from a chord operator, then its playable frequency vector is:

$$\boxed{\mathbf{f} = T_{\text{oct}}\,\mathbf{c}}.$$

The non-zero entries of $\mathbf{f}$ are passed to an audio synthesis function, where sinusoids (with added overtones) are generated for each frequency component.

2.5.4 Significance

- Harmony is computed purely in $\mathbb{R}^{12}$ independent of sound or octave.
- The transformation T_{oct} is the only step that introduces real frequencies.
- Playback on speakers becomes a final linear mapping from musical structure to audio data.

Thus, sound synthesis becomes the final linear transformation in the music generation pipeline.

2.5.5 Example: Mapping a C Major Chord to Frequencies

Consider the C major triad in pitch-class vector form:

$$\mathbf{c}_{\text{Cmaj}} = \mathbf{e}_1 + \mathbf{e}_5 + \mathbf{e}_8 = (1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0)^T,$$

corresponding to the pitch classes C, E, and G.

Frequency Assignment (3rd Octave)

Using equal temperament, the frequency of pitch class i in the 3rd octave is:

$$f_i = 440 \times 2^{\frac{i-10}{12}},$$

since A4 = 440 Hz and A3 corresponds to $440 \times 2^{-1} = 220$ Hz.

For the 3rd octave:

$$\begin{aligned} f_{\mathrm{C3}} &= 130.81 \text{ Hz}, \\ f_{\mathrm{E3}} &= 164.81 \text{ Hz}, \\ f_{\mathrm{G3}} &= 196.00 \text{ Hz}. \end{aligned}$$

Octave Transformation Matrix

The octave transformation matrix for the 3rd octave is therefore:

$$T_{\mathrm{oct},3} = \mathrm{diag}(130.81, 138.59, 146.83, 155.56, 164.81, 174.61, 185.00, 196.00, 207.65, 220.00, 233.08, 246.94).$$

Applying the Transformation

Multiplying the chord vector by this matrix yields its frequency representation:

$$\mathbf{f} = T_{\mathrm{oct},3}\, \mathbf{c}_{\mathrm{Cmaj}} = \begin{bmatrix} 130.81 \\ 0 \\ 0 \\ 0 \\ 164.81 \\ 0 \\ 0 \\ 196.00 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}.$$

Interpretation

The transformation selects only the frequencies corresponding to active pitch classes. The resulting non-zero components represent the playable tones of the C major chord in the 3rd octave:

$$\mathbf{f}_{\mathrm{Cmaj},3} = \{130.81,\ 164.81,\ 196.00\} \text{ Hz}.$$

These frequencies are then used for sound synthesis on speakers (using additive sinusoids with overtones).

2.6 Time Encoding and the Chord Engine

2.6.1 Time Encoding Overview

Once chord frequencies are obtained, sound must be generated for a certain duration. In music, duration is expressed in *beats* and controlled by the tempo (BPM: beats per minute). The time in seconds associated with b beats is:

$$t = \frac{60}{\mathrm{BPM}} \times b.$$

This scalar duration determines how long the generated frequency signal will be played on the speaker. Thus, time enters the system only as a multiplicative factor that stretches the waveform.

2.6.2 The Complete Chord Engine

Let:

- $\mathbf{e}_r$ be the root note basis vector,

- $M_K = \sum_{k \in K} S^k$ be the chord operator defined by interval set K,
- T_{oct} be the octave frequency transformation matrix,
- $\tau = \dfrac{60}{\text{BPM}} \cdot b$ be the time duration.

Then the playable chord frequency vector is:

$$\boxed{\mathbf{f} = T_{\text{oct}} \left(M_K \, \mathbf{e}_r \right)}.$$

Incorporating time encoding for sound synthesis yields the final playback operation:

$$\boxed{\mathbf{s}(t) = \text{Synth}\left(T_{\text{oct}} \left(\sum_{k \in K} S^k \mathbf{e}_r \right),\ \tau \right)}$$

Interpretation

- $\displaystyle\sum_{k \in K} S^k$ constructs the chord algebraically.
- T_{oct} maps the chord to real frequencies.
- τ controls duration via tempo.

Thus, harmony (via shift matrices), pitch realization (via frequency mapping), and time (via beat scaling) together form the complete linear-algebraic chord engine.

Chapter 3

Implementation (Python)

In this chapter, we implement the chord engine step by step, following the mathematical model developed earlier. The implementation is done in Python using NumPy for vector and matrix operations and a sound library for playback.

3.1 Implementation of Note Vectors

The first step is to encode each pitch class as a one-hot vector in $\mathbb{R}^{12}$. We define a mapping from note names (such as `"C"`, `"D#"`, `"A#"`) to indices $0, \ldots, 11$, and then create a length-12 NumPy vector with a single 1 at the corresponding index.

Pitch-Class Index Mapping

We fix the following index convention for the twelve pitch classes:

$$\mathrm{C} \to 0,\ \mathrm{C}^\sharp/\mathrm{D}^\flat \to 1,\ \mathrm{D} \to 2,\ \ldots,\ \mathrm{B} \to 11.$$

In Python, this is implemented as a dictionary:

```python
import numpy as np

NOTE_INDEX = {
    "C": 0,
    "C#": 1, "Db": 1,
    "D": 2,
    "D#": 3, "Eb": 3,
    "E": 4,
    "F": 5,
    "F#": 6, "Gb": 6,
    "G": 7,
    "G#": 8, "Ab": 8,
    "A": 9,
    "A#": 10, "Bb": 10,
    "B": 11
}
```

Note Vectors Using a DataFrame

To convert each note into its corresponding vector representation, I created a pandas DataFrame called `df_notes`. Each row stores a pitch name and its associated one-hot vector in $\mathbb{R}^{12}$. The implementation in the notebook is:

```
import numpy as np
import pandas as pd

PITCH_NAMES = ["C","C#","D","D#","E","F","F#","G","G#","A","A#","B"]
vectors = np.eye(12)

df_notes = pd.DataFrame({
    "Note": PITCH_NAMES,
    "Vector": [v for v in vectors]
})
```

The Out gave me a dataframe that stores vectors for every note

Table 3.1: One–Hot Vector Representation of Pitch Classes in $\mathbb{R}^{12}$

Index	**Note**	**Vector Representation**
0	C	[1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
1	C#	[0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
2	D	[0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
3	D#	[0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
4	E	[0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
5	F	[0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
6	F#	[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0]
7	G	[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0]
8	G#	[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0]
9	A	[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0]
10	A#	[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0]
11	B	[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0]

Accessing Note Vectors from the DataFrame

To make the note representation more practical, we retrieve one-hot vectors directly from the DataFrame `df_notes`. The function `get_note_vector` takes a note name as input (such as `"C"`, `"G#"`, or `"Bb"`) and returns its corresponding vector in $\mathbb{R}^{12}$.

The function:

- converts input to uppercase,
- handles the Unicode sharp symbol ("ˇ266F") by replacing it with "#",
- searches for the note in `df_notes`,
- returns its stored one-hot vector.

Python implementation:

```
def get_note_vector(note_name, df=df_notes):
    note = note_name.upper().replace("\u266F", "#")
    match = df[df["Note"].str.upper() == note]
    if match.empty:
        raise ValueError(f"Note '{note_name}' not found in DataFrame.")
    return match["Vector"].values[0]
```

This function allows us to retrieve any note as a 12D vector and acts as the base input for chord generation, transposition, and frequency mapping.

3.2 Implementation of Shift Matrices

The shift matrix represents transposition in music. A shift of k semitones is achieved by cyclically moving the pitch-class basis vectors by k positions. This operation is implemented in Python using `numpy.roll`, which rotates rows of the identity matrix.

```
def shift_matrix(k):
    return np.roll(np.eye(12), k, axis=0)
```

- `np.eye(12)` creates a 12×12 identity matrix.
- `np.roll(..., k, axis=0)` rotates the rows downward by k steps.
- The resulting matrix shifts any note vector by k semitones.
- Because the roll wraps cyclically, pitch classes remain modulo 12.

```
print(shift_matrix(5    ))
```

```
    [[0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0.],
    [0., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0.],
    [0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0.],
    [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 0.],
    [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1.],
    [1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
    [0., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
    [0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
    [0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0.],
    [0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0.],
    [0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0.],
    [0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0.]]
```

3.3 Chord Operator Generator

Chord Interval Definitions in Python

To construct chord operators programmatically, we first require the interval structure of each chord. In musical terms, a chord is defined by semitone distances above the root. For example, a major triad consists of the root (0), a major third (+4 semitones) and a perfect fifth (+7 semitones).

We store these interval recipes in a Python dictionary named `CHORD_STRUCTURES`. It organizes chords by type (major, minor, diminished, etc.) and by subtype (triad, seventh, suspended, etc.). Each entry is a list of semitone offsets that will later be transformed into shift matrices.

```
CHORD_STRUCTURES = {
    "major": {
        "triad": [0, 4, 7],         # major triad
        "7":     [0, 4, 7, 10],     # dominant 7
        "maj7":  [0, 4, 7, 11],     # major 7
    },
    "minor": {
        "triad": [0, 3, 7],         # minor triad
        "7":     [0, 3, 7, 10],     # minor 7
    },
    "diminished": {
        "triad": [0, 3, 6],         # dim triad
        "7":     [0, 3, 6, 9],      # fully dim 7
        "m7b5":  [0, 3, 6, 10],     # half-diminished (min7b5)
    },
    "augmented": {
        "triad": [0, 4, 8],         # augmented triad
    },
    "suspended": {
        "sus2":  [0, 2, 7],         # sus2 chord
        "sus4":  [0, 5, 7],         # sus4 chord
    },
    "power": {
        "5":     [0, 7],            # power chord (root + 5th)
    }
}
```

Usefulness in the Chord Engine

- Each interval set specifies which powers of the shift matrix will be summed to produce a chord operator.
- A general operator can be computed as:

  $$M_{\text{chord}} = \sum_{k \in K} S^k,$$

  where K is the list extracted from `CHORD_STRUCTURES`.
- This allows new chords to be added simply by updating the dictionary, with no changes to mathematical code.

Thus, `CHORD_STRUCTURES` acts as the declarative part of the chord engine, while the linear algebra code remains general and reusable.

Generating Chord Operators

The function `chord_op` constructs the linear operator that generates a chord from a root note. It selects the semitone structure from `CHORD_STRUCTURES`, and for each interval, adds the corresponding shift matrix to form:

$$M_{\text{chord}} = \sum_{k \in K} S^k.$$

```python
def chord_op(chord_type, subtype=None):
    t = chord_type.lower()
    st = subtype.lower() if isinstance(subtype, str) else None
    if st is None:
        if t in ("major", "minor", "diminished", "augmented"):
            st = "triad"
        elif t == "power":
            st = "5"
        else:
            raise ValueError(f"No default subtype for chord type '{chord_type}'
                ")
    if t not in CHORD_STRUCTURES:
        raise ValueError(f"Unknown chord type: '{chord_type}'")
    if st not in CHORD_STRUCTURES[t]:
        raise ValueError(f"Unknown subtype '{subtype}' for chord type '{
            chord_type}'")
    intervals = CHORD_STRUCTURES[t][st]
    M = np.zeros((12, 12), dtype=float)
    for k in intervals:
        M += shift_matrix(k % 12)

    return M
```

How the Function Works

- Reads semitone intervals for the requested chord from `CHORD_STRUCTURES`.
- Automatically assigns a default subtype (e.g., triad, sus, power chord).
- Creates an empty 12×12 matrix.
- Adds the appropriate shift matrices to form $M_{\text{chord}} = \sum S^k$.
- Returns the linear operator that can be multiplied with a root note vector.

This operator can now be applied to any note vector to construct its corresponding chord. Example output of this function is demonstrated as generating an operator for minor 7 version of a note here is the minor 7 operator-

```
[[1. 0. 0. 0. 0. 1. 0. 0. 1. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 1. 0. 0. 1. 0. 0.]
 [0. 0. 1. 0. 0. 0. 0. 1. 0. 0. 1. 0.]
 [0. 0. 0. 1. 0. 0. 0. 0. 1. 0. 0. 1.]
 [1. 0. 0. 0. 1. 0. 0. 0. 0. 1. 0. 0.]
 [0. 1. 0. 0. 0. 1. 0. 0. 0. 0. 1. 0.]
 [0. 0. 1. 0. 0. 0. 1. 0. 0. 0. 0. 1.]
 [1. 0. 0. 1. 0. 0. 0. 1. 0. 0. 0. 0.]
 [0. 1. 0. 0. 1. 0. 0. 0. 1. 0. 0. 0.]
 [0. 0. 1. 0. 0. 1. 0. 0. 0. 1. 0. 0.]
 [0. 0. 0. 1. 0. 0. 1. 0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 1. 0. 0. 1. 0. 0. 0. 1.]]
```

Generating Chord Vectors

Once the chord operator M_{chord} is constructed, a chord in $\mathbb{R}^{12}$ is obtained by multiplying this operator with the root note vector. If $\mathbf{v}_{\text{root}}$ is the one–hot vector for the root pitch class, then the chord vector is:

$$\mathbf{c} = M_{\text{chord}}\,\mathbf{v}_{\text{root}}.$$

In Python, this is implemented using simple martix multiplication in numpy:

```
    note_vec    = get_note_vector(note_name)
    chord_vec   = chord_op(chord_type, subtype) @ note_vec
```

Example: C Major Triad

```
cmaj_vec = chord_op("major", "triad")@get_note_vector("c")
print(cmaj_vec)
```

This prints a vector with ones at the positions corresponding to C, E and G, and zeros elsewhere, representing the C major chord in $\mathbb{R}^{12}$.

3.4 Octave Transformation Matrix

To convert chord vectors into physical frequencies, we must assign a frequency (in Hertz) to each of the twelve pitch classes. This is done using a diagonal matrix, where each diagonal entry corresponds to the frequency of that note in a specific octave.

We first define the frequency matrix for the 4th octave (where middle C lies):

```
C4 = 261.63
freq_4 = np.array([C4 * 2**(k/12) for k in range(12)])  # C4..B4
T4 = np.diag(freq_4)
```

The list comprehension generates the twelve equal–tempered frequencies:

$$f_k = f_{\text{C4}} \cdot 2^{\frac{k}{12}}, \qquad k = 0, 1, \dots, 11.$$

Scaling to Other Octaves

Every octave doubles or halves frequencies relative to the 4th octave. Hence, a general transformation matrix for octave n is created by scaling T_4 by:

$$2^{(n-4)}.$$

Python implementation:

```
def generate_transform(octave):
    factor = 2**(octave - 4)
    return factor * T4
```

Example

To obtain frequencies for the 3rd octave, we compute:

$$T_3 = 2^{(3-4)} \cdot T_4 = \frac{1}{2}T_4.$$

In Python:

```python
T3 = generate_transform(3)
print(np.diag(T3)[:5])  # first 5 pitch frequencies in octave 3
```

This will return approximately:

$$[130.81,\ 138.59,\ 146.83,\ 155.56,\ 164.81] \text{ Hz},$$

representing C3, C#3, D3, D#3 and E3, respectively.

Thus, the octave transformation scales the frequency space while the chord operator determines *which* notes are active.

3.5 Mapping Chord Vectors to Frequencies

Once a chord vector $\mathbf{c} \in \mathbb{R}^{12}$ is computed using the chord operator, it must be converted into real sound frequencies. This is achieved by multiplying the chord vector with the octave transformation matrix:

$$\mathbf{f} = T_{\text{oct}}\,\mathbf{c},$$

where T_{oct} is the diagonal matrix returned by `generate_transform(octave)`.

The non-zero entries of $\mathbf{f}$ represent the frequencies (in Hertz) of the notes inside the chord.

```python
freq_vec   = generate_transform(octave) @ chord_vec
```

Example: Frequencies of C Major in the 3rd Octave

```python
freq_vec   = generate_transform(3) @ chord_vec
```

This produces approximately:

$$\{130.81,\ 164.81,\ 196.00\} \text{ Hz},$$

corresponding to the notes C3, E3 and G3 in the C major triad.

Thus, octave scaling determines *how high or low* the notes sound, while the chord operator determines *which notes* are present.

3.6 Complete Chord Playback Function

The final step of the chord engine combines note vector extraction, chord operator application, octave transformation, and audio synthesis into a single function. The function `play_chord_type` takes a root note, an octave, and a chord type, and produces the audible chord using the library `sounddevice`.

```python
def play_chord_type(note_name, octave, chord_type, subtype):
    note_vec   = get_note_vector(note_name)
    chord_vec  = chord_op(chord_type, subtype) @ note_vec
    freq_vec   = generate_transform(octave) @ chord_vec
    play_chord(freq_vec)
```

How the Function Works

- Converts the input note to a one–hot vector in $\mathbb{R}^{12}$.

- Applies the chord operator to generate the chord vector.
- Maps the chord vector to actual frequencies using the octave matrix.
- Sends the resulting frequencies to the synthesis function `play_chord`.

Mathematically, the function performs:

$$\text{Synth}\big(T_{\text{oct}}\big(M_{\text{chord}}\mathbf{e}_{\text{root}}\big)\big),$$

thereby converting the pure algebraic representation into audible sound.

3.7 Progression and Tempo

So far, we have seen how to generate and play a single chord. To play a complete musical passage, we need to handle a *sequence* of chords together with their durations. This is done by storing all chord vectors as columns of a matrix and associating a beat duration with each chord.

Building a Chord Progression Matrix

A chord progression is represented in Python as a list of tuples:

$$(\text{root note},\ \text{chord type},\ \text{subtype},\ \text{beats}).$$

For example:

`[("C","major","triad", 2), ("F","major","triad", 2),...].`

The function `build_progression_with_beats` converts this list into:

- a matrix $C \in \mathbb{R}^{12\times N}$ whose columns are chord vectors,
- a beat vector $b \in \mathbb{R}^{N}$ storing the duration (in beats) of each chord.

```python
def build_progression_with_beats(progression):
    chord_cols = []
    beats = []

    for (note, ctype, sub, b) in progression:
        root_vec = get_note_vector(note)
        M = chord_op(ctype, sub)
        chord_vec = M @ root_vec
        chord_cols.append(chord_vec)
        beats.append(b)

    C = np.column_stack(chord_cols)        # 12 x N chord matrix
    beats = np.array(beats, dtype=float)   # length-N beat vector
    return C, beats
```

Mathematically,

$$C = [\,\mathbf{c}_1\ \mathbf{c}_2\ \dots\ \mathbf{c}_N\,], \qquad b = (b_1, \dots, b_N),$$

where each $\mathbf{c}_j$ is a chord vector and b_j is its duration in beats.

Tempo and Playback of the Progression

To play the progression, we first map all chords to frequencies in a chosen octave using the transformation matrix T_{oct}:

$$F = T_{\text{oct}}\, C,$$

so the j-th column $\mathbf{f}_j$ contains the frequencies of the j-th chord.

The tempo is given in beats per minute (BPM). Each chord's time (in seconds) is computed as:

$$t_j = b_j \cdot \frac{60}{\text{BPM}}.$$

The following function implements this logic:

```
def play_progression_tempo(progression, octave, bpm):
    C, beats = build_progression_with_beats(progression)
    T = generate_transform(octave)
    F = T @ C

    seconds_per_beat = 60.0 / float(bpm)
    N = C.shape[1]

    for j in range(N):
        freq_vec = F[:, j]                   # j-th chord frequencies
        dur_seconds = beats[j] * seconds_per_beat
        play_chord(freq_vec, duration=dur_seconds)
```

In summary:

- Each chord is a column in the matrix C.
- All chords are mapped to the frequency domain by $F = T_{\text{oct}}C$.
- The beat vector and BPM together determine the time spent on each column.
- The progression is played by iterating over columns of F and calling `play_chord`.

This completes the chord engine: linear algebra handles harmony and structure, while tempo and beats control the timing of chord playback.

Chapter 4

Results and Demonstration

Example: Chord Progression for a Popular Song (“Jeena Jeena”)

To demonstrate the complete chord engine in a real musical context, we encode a chord progression inspired by the song *“Jeena Jeena”*. Each tuple stores:

$$(\text{root note, chord type, subtype, beats}).$$

```
jeena_jeena = [
    # F                  Dm
    # Sacchi si hai yeh tareefein
    ("F", "major", "triad", 4),
    ("D", "minor", "triad", 4),

    # F                C
    # dil se jo maine kari hain
    ("F", "major", "triad", 4),
    ("C", "major", "triad", 4),

    # Dm         Am
    # Jo tu mila toh saji hai
    ("D", "minor", "triad", 4),
    ("A", "minor", "triad", 4),

    # Gm          C
    # duniya meri humdum
    ("G", "minor", "triad", 4),
    ("C", "major", "triad", 4),
]

play_progression_tempo(jeena_jeena, octave=4, bpm=105)
```

Explanation

- Each chord is specified by its root, quality (major/minor), subtype (`"triad"`) and duration (4 beats).
- The list `jeena_jeena` is passed to `play_progression_tempo`, which:
    - converts each chord to a column of the chord matrix C,

- maps C to the frequency matrix $F = T_{\text{oct}}C$ for octave 4,
- computes each chord duration from the tempo (`bpm = 105`),
- plays the chords sequentially using `play_chord`.

This example shows how a familiar progression can be described entirely by chord parameters and beats, while all underlying note generation, transposition, frequency mapping and timing are handled by the linear algebra chord engine.

Conclusion

In this work, we demonstrated how musical harmony can be generated and performed using the principles of linear algebra. By representing pitch classes as basis vectors in $\mathbb{R}^{12}$, and defining chords as linear operators constructed from cyclic shift matrices, we obtained a purely algebraic model of chord construction and transposition. Mapping these vectors to physical frequencies through octave transformation matrices connected the abstract mathematical structure to audible sound. Finally, by encoding progressions as matrices with time durations tied to tempo, we were able to synthesize complete musical passages, such as the progression from "Jeena Jeena."

This framework shows that musical creation can be expressed as a sequence of matrix operations, where harmony, frequency, and time are unified through linear transformations. The result is a systematic and extendable "chord engine" that not only explains music mathematically but also produces real sound from algebraic expressions.