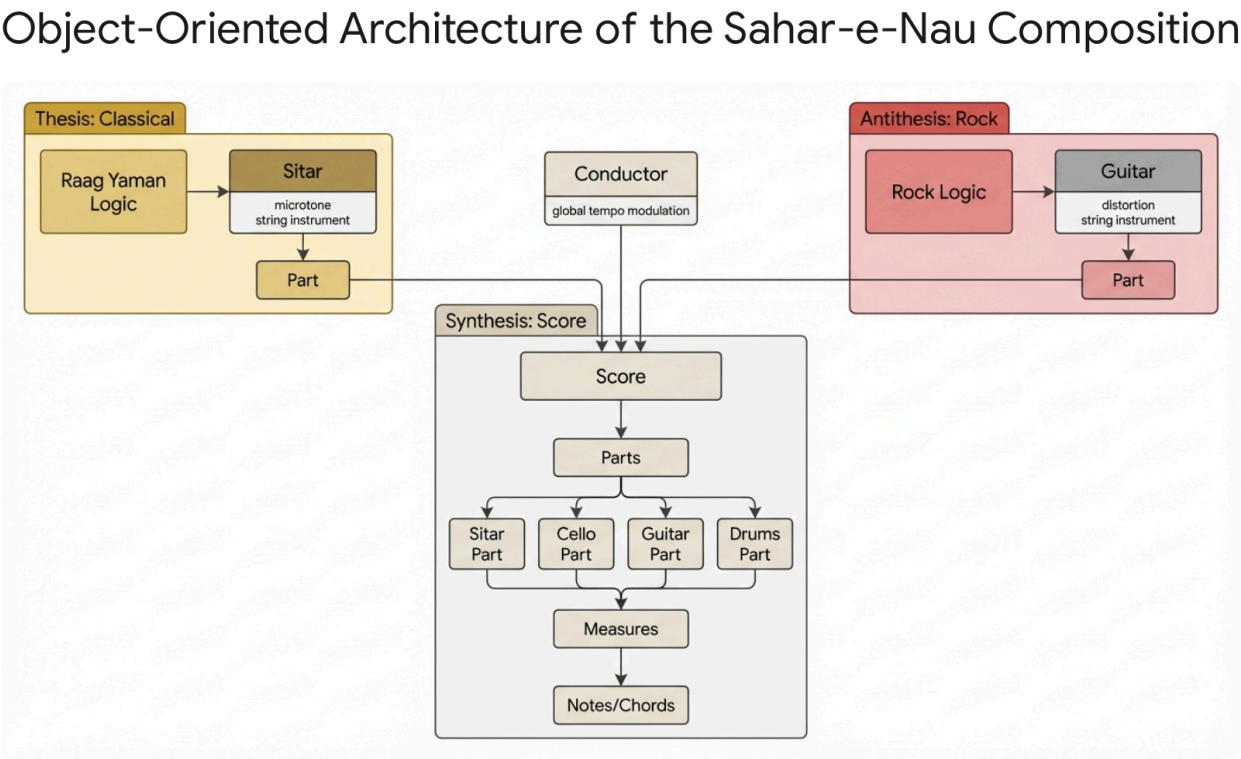
Computational Orchestration of *Sahar-e-Nau*: A Music21 Implementation Framework

1. Introduction: The Algorithmic Architecture of Dissent

The intersection of computational musicology and political expression offers a unique vantage point for analyzing the artistic trajectory of Faiz Ahmed Faiz. The proposed fusion, titled *Sahar-e-Nau* (The New Dawn), is not merely a juxtaposition of two seminal works—the romantic *ghazal* "Mujh Se Pehli Si Muhabbat" and the revolutionary anthem "Umeed-e-Sehar"—but a deliberate collision of two distinct musicological, philosophical, and temporal systems.[1] One is rooted in the cyclical, introspective traditions of North Indian classical music (Hindustani Sangeet), specifically the heptatonic yearning of *Raag Yaman*; the other is grounded in the linear, driving teleology of Western rock protest music, characterized by binary rhythms and functional harmony.[1] Translating this fusion into a symbolic representation using Python's music21 library requires a rigorous translation of cultural and emotional intent into object-oriented programming structures. It demands that we treat musical parameters not as static data points, but as dynamic variables in a complex system of cultural friction.

This report serves as a comprehensive technical and theoretical blueprint for implementing the instrumental score of *Sahar-e-Nau*. By bypassing the vocal lines to focus exclusively on the orchestral skeleton—the interaction between the Sitar, Cello, Electric Guitar, and Percussion—we expose the structural tension that underpins the fusion's narrative arc. The core challenge lies in programmatically reconciling the *Tivra Ma* (Sharp Fourth) of Raag Yaman with the functional harmony of the Western Major scale, and bridging the fluid *rubato* of the *ghazal* with the quantized grid of the rock anthem.[1] We must write code that does not simply play notes, but enacts the "dialectic of love and revolution" inherent in Faiz's poetry.



A hierarchical view of the Python class structure designed for the composition. The diagram illustrates how the 'Thesis' module (Raag Yaman/Sitar) and 'Antithesis' module (Rock/Guitar) feed into the central 'Synthesis' Stream, managed by a global 'Conductor' object that handles tempo modulation.

The music21 library, a toolkit for computer-aided musicology developed at MIT, provides the ideal environment for this undertaking. Its Stream architecture allows for the hierarchical organization of music (Score > Part > Measure > Note) [2], while its sophisticated Pitch and Duration objects allow for the manipulation of microtonal inflections and complex tuplets essential for the "Eastern" sections of the piece.[3] This report will proceed bar-by-bar, dissecting the theoretical physics of the fusion and providing the Python logic necessary to render it. We will explore how to instantiate specific instruments like the Sitar and Tabla within a framework designed for Western notation, how to use Duration objects to simulate the "lilt" of *Keherwa* taal, and how to use Key and Scale objects to enforce the harmonic "fracture" that defines the piece's middle section.

2. The Computational Environment: Music21 Object Model

To implement *Sahar-e-Nau*, we must first establish a robust computational environment. music21 operates on an object-oriented model where musical elements are treated as objects contained within hierarchical lists called Streams.[5] Understanding this hierarchy is crucial for

managing the complex interplay of voices in our fusion.

2.1 The Stream Hierarchy: Score, Part, Measure

In music21, a Stream is the fundamental container. It acts as a superclass for Score, Part, and Measure objects.[2] For our orchestral score, we will instantiate a stream.Score object as the root container. This score will contain multiple stream.Part objects, each representing a distinct instrument (Sitar, Cello, Electric Guitar, Drums). Inside each Part, time is organized into stream.Measure objects, which finally contain the atomic note.Note, chord.Chord, and note.Rest objects.[6]

The power of this architecture lies in its flexibility. A note does not just exist in isolation; it has an activeSite property that links it to the measure and part it belongs to, allowing for context-aware analysis and manipulation.[7] For *Sahar-e-Nau*, this allows us to programmatically apply transformations—such as transposition or rhythmic augmentation—to specific parts without affecting the whole.

Python

```
from music21 import *

def initialize_score():
    """
    Initializes the master Score object with metadata and empty Parts.
    """
    score = stream.Score()
    score.metadata = metadata.Metadata()
    score.metadata.title = "Sahar-e-Nau: Symphony of the Awakening"
    score.metadata.composer = "Faiz Fusion Project"

    # Initialize Parts
    parts = {
        'Sitar': stream.Part(),
        'Cello': stream.Part(),
        'Guitar': stream.Part(),
        'Drums': stream.Part()
    }

    # Assign Instrument Classes (See Section 5 for details)
    parts.insert(0, instrument.Sitar())
```

```
    parts['Cello'].insert(0, instrument.Violoncello())
    parts['Guitar'].insert(0, instrument.ElectricGuitar())
    parts.insert(0, instrument.DrumKit()) # Placeholder for custom mapping

    for p_name, part in parts.items():
        part.id = p_name
        score.insert(0, part)

    return score, parts
```

2.2 Temporal Management: Offsets and Durations

Time in music21 is measured in "quarter lengths" (QL). A quarter note has a QL of 1.0, a half note 2.0, and an eighth note 0.5.[4] This floating-point representation of time is critical for our fusion because it allows for the precise calculation of complex tuplets required for the *Keherwa* rhythm. Unlike a DAW (Digital Audio Workstation) that relies on samples or milliseconds, music21 relies on symbolic duration. This means we can mathematically model the "swing" of the *ghazal* by manipulating the QL ratios (e.g., using duration.Tuplet objects to create a 2:1 swing ratio) before the music is ever rendered to audio.[4]

3. Theoretical Modeling: The Variable Geometry of Pitch

Before instantiating a single note, we must define the tonal universe in which our objects will exist. The friction in *Sahar-e-Nau* arises from the clash between two scales: the modal purity of *Raag Yaman* and the functional harmony of the Western Major scale.[1]

3.1 The *Ma* Paradox: Defining Raag Yaman in Python

Raag Yaman is a heptatonic (*Sampurna*) raga that corresponds roughly to the Lydian mode in Western theory. Its defining characteristic is the **Tivra Ma** (Sharp Fourth, F# in the key of C, or G# in the key of D).[1] In Indian classical theory, this note is not merely a pitch; it is a vector of emotion, often approached via a *Meend* (gliding slide) from the Fifth (*Pa*) or the Third (*Ga*), creating a sense of suspended yearning.

In music21, we can model this scale using the scale module. While music21 has built-in support for Western scales, raga implementation requires constructing a custom scale object or valid pitch list. The "Ma Paradox" refers to the conflict between this *Tivra Ma* (G#) used in the nostalgic section and the *Shuddha Ma* (G Natural) used in the revolutionary section.[1]

The Code Logic for Raag Yaman

We define a custom function to generate the Yaman scale over a specified range. Although

strict classical performance omits the Root (*Sa*) and Fifth (*Pa*) in the ascent (*Aroh*), the instrumental *ghazal* style (*Sugam Sangeet*) often employs the full heptatonic scale for melodic fluidity.

Python

```
def get_yaman_scale(root_note='D4'):
    """
    Returns a list of pitches for Raag Yaman based on the root.
    Formula: R (Major 2nd), G (Major 3rd), M# (Aug 4th), P (Perfect 5th),
             D (Major 6th), N (Major 7th).
    """
    tonic = pitch.Pitch(root_note)
    # Intervals from Tonic: Maj2, Maj3, Aug4, Perf5, Maj6, Maj7
    # Note: 'A4' represents Augmented 4th (Tivra Ma)
    intervals = ['M2', 'M3', 'A4', 'P5', 'M6', 'M7']

    yaman_pitches = [tonic]
    for int_str in intervals:
        p = tonic.transpose(int_str)
        yaman_pitches.append(p)

    return yaman_pitches

# Example: Generating Yaman in D
yaman_d = get_yaman_scale('D4')
# Output:
```

This list yaman_d becomes our tonal palette. Any algorithmic generation for the Sitar part must strictly draw from this list. The music21.pitch.Pitch object handles the accidental G#4 automatically, ensuring that the *Tivra Ma* is correctly represented in any output format (MusicXML, MIDI).[3]

3.2 The Minor IV: The Harmonic Anchor of Revolution

The second theoretical pillar is the chord progression of *Umeed-e-Sehar*. The anthem uses a standard D Major progression but borrows the **Minor IV (Gm)** from the parallel minor (D Minor).[1] This Gm chord (G - Bb - D) introduces two notes that are alien to Raag Yaman:

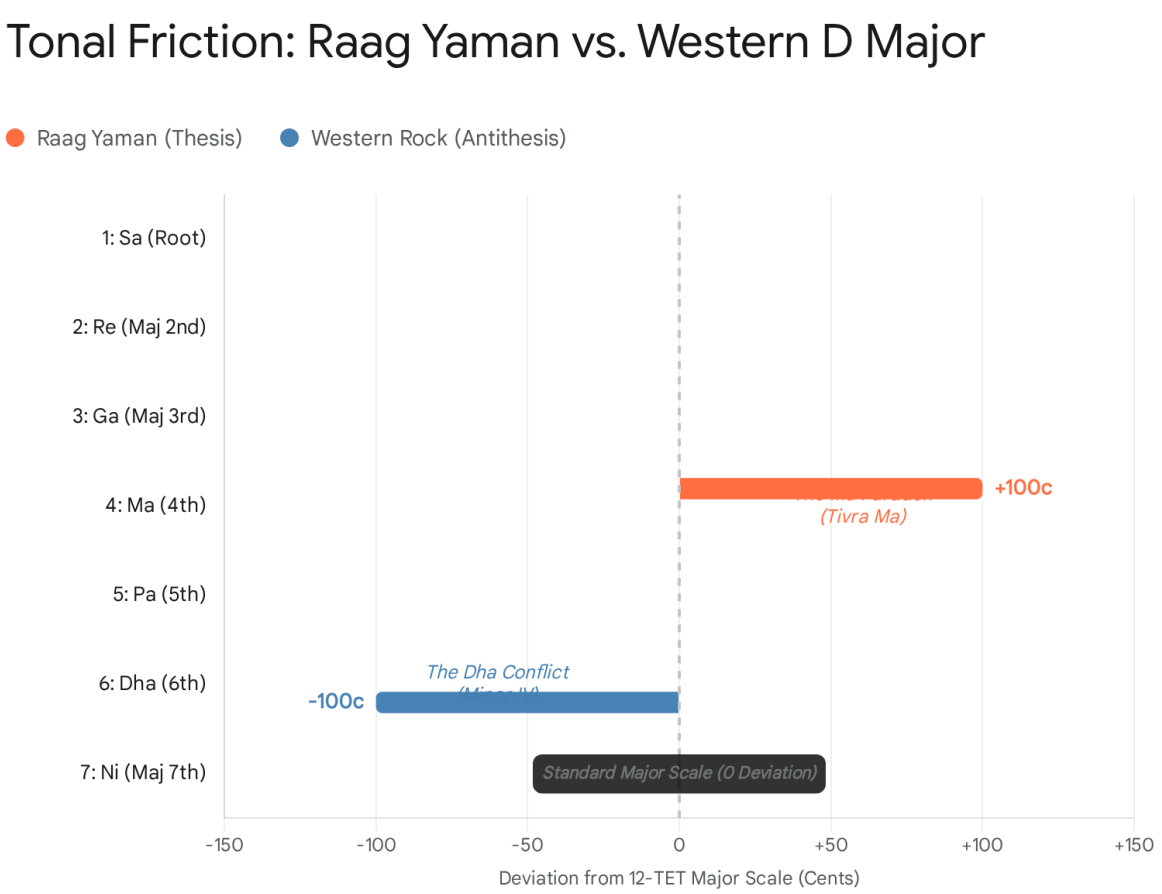
1. **G Natural (Shuddha Ma):** Conflicts with Yaman's G# (Tivra Ma).
2. **Bb (Komal Dha):** Conflicts with Yaman's B Natural (Shuddha Dha).[1]

This **"Dha Conflict"** is the emotional pivot of the piece. The Bb acts as a "wounding" of the scale, signaling the intrusion of pain into the romantic worldview. In music21, we define these chords explicitly to ensure the correct spelling and voice leading.

Python

```
# Define the Revolutionary Progression
chord_progression = {
    'I': chord.Chord(),   # D Major
    'V': chord.Chord(['A2', 'C#3', 'E3']),   # A Major
    'IV': chord.Chord(),   # G Major (Standard)
    'iv': chord.Chord(),  # G Minor (The "Hope" Chord)
}
```

The computational challenge is to construct a "Poly-Scalar" environment where the Sitar continues to reference Yaman (using B Natural and G#) while the Rhythm Guitar forces the D Major / Minor IV context (using Bb and G Natural). This cross-relation is not a mistake; it is the sonic representation of the "fracture" described in the fusion document.



Comparison of the Pitch Classes used in the Thesis (Yaman) and Antithesis (Rock) sections. The bars extend to the specific cent deviation from 12-TET. Note the +100 cent deviation at the 4th degree (The 'Ma Paradox') and the -100 cent deviation at the 6th degree (The 'Dha Conflict').

Data sources: Faiz Fusion: Classical to Revolutionary, Music21 Scale, Music21 Pitch

4. The Rhythmic Architecture: From Fluidity to Grid

The rhythmic transition in *Sahar-e-Nau* mirrors the philosophical shift from the circular time of the *ghazal* to the linear, marching time of the revolution. We are moving from a *Keherwa* beat (8 beats, often swung/loping) to a Rock 4/4 beat (8 beats, straight, binary).

4.1 Implementation of *Keherwa* using Tuplets

Keherwa is theoretically a 4+4 cycle, but in the semi-classical *ghazal* style, it is rarely played as straight eighth notes. It possesses a characteristic "lilt" or swing, often described as a *Laggi*. To capture this in music21 without relying on vague "humanize" functions, we must use **Tuplets**. By dividing the beat into triplets (12/8 compound time) rather than duplets, we can approximate this swing.[4]

We can code a basic *Keherwa* pattern using note.Note objects. The pattern *Dha - Ge - Na - Tin | Na - Ka - Dhin - Na* can be mapped to specific durations. To achieve the "lilt," we can use a ratio where the first 8th note of a pair is slightly longer than the second, or map the entire cycle to a triplet grid.

Python

```
def create_keherwa_cycle():
    """
    Creates one bar of Keherwa Theka (8 beats) with a swung feel.
    Pattern: Dha - Ge - Na - Tin | Na - Ka - Dhin - Na
    Represented using triplet subdivisions for the 'lilt'.
    """
    m = stream.Measure()
    m.timeSignature = meter.TimeSignature('4/4')

    # Mapping Tabla bols to MIDI (See Section 5.3 for details)
    # 36=Kick(Ge), 38=Snare(Na/Ka), 45=LowTom(Dha/Dhin)
    # Using triplets: Beat is divided into 3 parts.
    # 'Swing' feel: First note = 2/3 beat, Second note = 1/3 beat.

    theka_pattern =

    for midi_val, q_len in theka_pattern:
        if midi_val is None:
            n = note.Rest(quarterLength=q_len)
        else:
            if midi_val > 100: # Combo stroke
                n = note.Note(36, quarterLength=q_len) # Simplified
            else:
                n = note.Note(midi_val, quarterLength=q_len)

        # Explicitly define tuplet to ensure correct notation/playback
        t = duration.Tuplet(3, 2)
        n.duration.tuplets = (t,)
        m.append(n)

    return m
```

4.2 Metric Modulation Strategy

The fusion document calls for a transition from this fluid style to a "gritty march." To achieve this smoothly, we employ a **Metric Modulation**.[1] This is a technique where a subdivision of the previous tempo becomes the main beat of the new tempo.

The Math: If the *Keherwa* section is at 72 BPM, and we are using triplet eighths, the speed of the triplets is $72 \times 3 = 216$ pulses per minute. In the Rock section, we want to straighten this out. If we equate the triplet eighth note of the old tempo to the straight eighth note of the new tempo, the new tempo becomes:

$$NewBPM = OldBPM \times 1.5$$

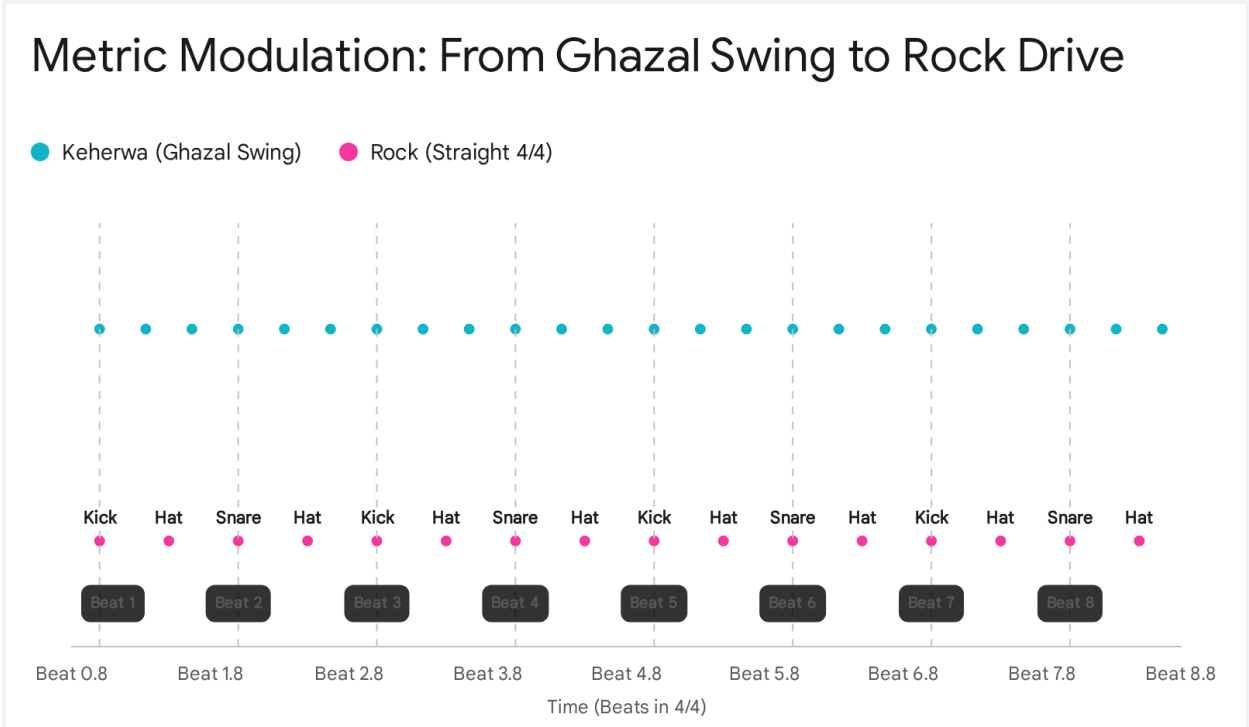
$$72 \times 1.5 = 108 \text{ BPM}$$

This results in a perceptible increase in energy and urgency—the "march" of the revolution. We implement this in music21 using tempo.MetronomeMark objects at specific offsets.[9]

Python

```
# Transition Logic
transition_stream = stream.Stream()
#... Ghazal measures...
mm1 = tempo.MetronomeMark(number=72)
transition_stream.append(mm1)

#... Transition measures...
# Insert Tempo Change for the Rock section
mm2 = tempo.MetronomeMark(number=108) # 72 * 1.5
transition_stream.append(mm2)
```



Timeline comparing the note onsets of the Keherwa Theka (Top) versus the Rock Beat (Bottom). The transition point (Bar 33) shows how the triplet subdivision of the Ghazal becomes the straight eighth-note pulse of the Anthem, resulting in a 1.5x tempo acceleration.

Data sources: Faiz Fusion: Classical to Revolutionary, Music21 Documentation, Reddit Music Theory

5. Instrumentation and Timbre: Object-Oriented Organology

music21 allows us to assign specific instruments to Stream parts using classes from the instrument module.[10] However, the library's default instrument definitions are mapped to General MIDI (GM) program numbers, which are optimized for Western orchestration. This poses significant challenges for accurately representing the non-Western instruments required for *Sahar-e-Nau*.

5.1 The Sitar Class: Overcoming GM Limitations

The General MIDI standard assigns Program 105 to "Sitar". In many basic MIDI synthesizers, this sound is thin and monophonic, completely lacking the *Jwari* (buzz) and the sympathetic resonance (*Tarab*) that define the instrument's sonic signature.[10] A standard instrument.Sitar() object in music21 will trigger this patch, but it will fail to convey the "Golden Cage" atmosphere.

Computational Strategy: We will simulate the acoustic complexity of the Sitar by instantiating **two parallel parts**:

1. **Melody Part (sitar_melody):** Plays the primary raga lines using instrument.Sitar().
2. **Shadow Drone Part (sitar_drone):** A secondary part that plays a continuous, low-velocity drone of the open strings (Sa and Pa, i.e., D and A). This mimics the sympathetic strings vibrating in response to the main melody. We can use the repeatAppend method to create this drone efficiently without deep-copying thousands of note objects manually.[11]

Python

```
# Main Melody Part
sitar_melody = stream.Part()
sitar_melody.insert(0, instrument.Sitar())

# Shadow Drone Part (Simulating Tarab strings)
sitar_drone = stream.Part()
sitar_drone.insert(0, instrument.Sitar())
# Logic: Add long-held D3 and A3 notes at low velocity (pp)
drone_root = note.Note('D3')
drone_root.volume.velocity = 30 # Pianissimo
drone_root.quarterLength = 100 # Long duration
sitar_drone.insert(0, drone_root)
```

5.2 The Cello: The Bridge Between Worlds

The Cello serves a dual function in this score. In the "Noor Jehan" section, it acts as a surrogate for the *Sarangi*, the traditional bowed instrument of the *ghazal*, mirroring the vocal line with fluid slides. In the "Laal" section, it transforms into a rock bass, locking in with the kick drum's grid.

To distinguish these roles programmatically, we will utilize articulations objects.[12]

- **Role 1 (Sarangi Substitute):** Notes will be assigned articulations.Legato and potentially pitch.microtone adjustments to simulate the fluid intonation. We may also use the SulPonticello articulation class if we want to evoke the "glassy," metallic sound of the Sarangi's sympathetic strings, a technique mentioned in the fusion document for the "Fracture" section.[1]
- **Role 2 (Rock Bass):** Notes will be assigned articulations.Marcato or articulations.Staccato to provide the punchy, detached articulation required for the rock

groove.

5.3 The Rhythm Section: Mapping the Tabla to MIDI

Perhaps the most critical deficiency in standard MIDI maps is the lack of a "Tabla" kit. A standard Drum Kit (MIDI Channel 10) maps keys to Kick, Snare, Hats, etc..[13] To score the *Keherwa* rhythm, we must create a **semantic mapping** that translates Tabla strokes (*bols*) to the closest approximations on a Western kit.[14]

The "Sahar-e-Nau" Percussion Map:

Tabla Stroke (Bol)	Description	Western Kit Approximation	MIDI Note	Reasoning
Ge / Ghe	Resonant Bass	Acoustic Bass Drum	35	Provides the low-end thud of the *Bayan* (left drum).
Na / Ta	Sharp Rim Sound	Side Stick / Rimshot	37 / 31	Mimics the sharp, metallic crack of the *Dayan* rim.
Tin	Soft Resonant	Low Tom / Snare (Soft)	45 / 38	Captures the mid-range resonance.
Dha	Bass + Rim (Ge + Na)	Bass Drum + Side Stick	35 + 37	The defining "beat" stroke; requires a chordal hit.
Dhin	Bass + Soft (Ge + Tin)	Bass Drum + Low Tom	35 + 45	The softer, resonant

				downbeat.
Ka / Ke	Flat Slap	High Hat Pedal / Dead Stroke	44	A muted, non-resonant slap.

By defining a dictionary in Python that maps these *bols* to their MIDI integers, we can write our code using the language of Tabla (add_stroke('Dha')) while generating valid MIDI output (Note(35) + Note(37)).

Python

```
tabla_map = {
    'Dha': ,
    'Ge': ,
    'Na': ,
    'Tin': ,
    'Dhin': ,
    'Ka':
}
```

This mapping allows us to code "Tabla" patterns using standard Note and Chord objects that music21 can export to MIDI, ensuring that the rhythm section functions correctly even without specialized sampling plugins.

6. Step-by-Step Construction: The Code Narrative

We will now assemble the piece, movement by movement, utilizing the classes and definitions established above.

6.1 Movement I: The Golden Cage (0:00 – 02:15)

This section establishes the "Thesis"—the beautiful but stagnant past of *Mujh Se Pehli Si Muhabbat*. The texture is sparse, dominated by the Sitar's *Alaap* (unmetered improvisation) and the Cello's drone.

Algorithmic Alaap Generation: Instead of hard-coding a static melody, we can utilize a

stochastic process (Random Walk) constrained by the rules of *Raag Yaman*. This ensures that every execution of the code generates a unique, yet theoretically correct, *Alaap*. We use weighted probabilities to favor the *Vaadi* (Dominant note, Ga/E) and *Samvaadi* (Sub-dominant, Ni/B), reflecting the raga's hierarchy.[3]

Python

```
import random

def generate_alaap(duration_quarters=32):
    """
    Generates a free-form Alaap in Raag Yaman.
    """
    alaap_stream = stream.Part()
    alaap_stream.insert(0, instrument.Sitar())

    # Yaman Pitch Set (focusing on middle register)
    # Weights favor the Vaadi (Third - G/E in D Major) and Samvaadi (Seventh - N/C#)
    # Note: For D Major root: G=F#, N=C#
    scale_degrees =
    weights =       [0.1,   0.2,  0.3,   0.1,   0.1,  0.1,  0.05,  0.05]

    current_offset = 0.0

    while current_offset < duration_quarters:
        # Choose a pitch based on weights
        p_str = random.choices(scale_degrees, weights=weights, k=1)

        # Choose a long, flowing duration (Rubato feel)
        dur = random.choice([2.0, 3.0, 4.0])

        n = note.Note(p_str)
        n.quarterLength = dur

        # Add a 'Meend' simulation (Pitch Bend) to Tivra Ma (G#)
        if 'G#' in p_str:
            # We can't render fluid slides easily, but we can add an expression
            # or a microtonal inflection to signal the intent.
            n.pitch.microtone = pitch.Microtone(10) # Sharpen by 10 cents
```

```
        alaap_stream.insert(current_offset, n)
        current_offset += dur

    return alaap_stream
```

6.2 Movement II: The Fracture (02:15 – 03:00)

Here, the "Turn" (*Gurez*) occurs. The poet's gaze shifts from the beloved to the suffering of the world. Musically, this is the moment of maximum dissonance. The fusion document specifies a clash between the "Silk" of the past and the "Pus" of reality.[1] We will implement this via the **Tritone Clash**.

The Sitar, still stuck in the dream of Yaman, plays the **G# (Tivra Ma)**. Simultaneously, we introduce a new textural element—perhaps a synthesized pad or the Cello playing *Sul Ponticello*—insisting on the **G Natural (Shuddha Ma)**. This interval (G - G#) is a minor second, highly dissonant, creating an acoustic "beating" effect that mimics the "siren" or "furnace" imagery.

Coding the Clash:

We will create a loop where these two notes alternate rapidly or sound simultaneously.

Python

```
def generate_fracture_texture():
    fracture_stream = stream.Part()
    fracture_stream.insert(0, instrument.Violoncello())

    # Rapid 16th note alternation creates a 'shimmering' dissonance
    for i in range(16): # 4 bars of 4/4
        # Note 1: The "Old" Love (G#)
        n1 = note.Note('G#3')
        n1.quarterLength = 0.25

        # Note 2: The "New" Reality (G Natural)
        n2 = note.Note('G3')
        n2.quarterLength = 0.25

        # Sul Ponticello articulation for 'metallic' sound
        sp = articulations.SulPonticello()
        n1.articulations.append(sp)
```

```
        n2.articulations.append(sp)

        # Dynamic Build: Crescendo from p to f
        velocity = 60 + (i * 2)
        n1.volume.velocity = velocity
        n2.volume.velocity = velocity

        fracture_stream.append(n1)
        fracture_stream.append(n2)

    return fracture_stream
```

6.3 Movement III: The March (03:00 – 05:00)

The *Umeed-e-Sehar* anthem begins. This is the "Antithesis." The instrumentation shifts decisively to the Rock Kit and Electric Guitar. The harmony moves from the ambiguous drone to the defined chord progression of the revolution.

The Power Chord Generator:

Rock music relies on "Power Chords" (Root + Fifth, no Third). This makes the harmony ambiguous and forceful. We can write a helper function to generate these efficient chord objects.

Python

```
def make_power_chord(root_name, duration=1.0):
    """
    Creates a Power Chord (Root + 5th)
    """
    r = note.Note(root_name)
    fifth = r.transpose('P5') # Transpose up a Perfect Fifth

    # Create chord object
    c = chord.Chord([r, fifth])
    c.quarterLength = duration
    return c

# Constructing the Riff: D -> A -> G -> Gm
riff_stream = stream.Part()
riff_stream.insert(0, instrument.ElectricGuitar())
```

```
# Bar 1: D Major (Power)
riff_stream.append(make_power_chord('D3', 4.0))

# Bar 2: A Major (Power)
riff_stream.append(make_power_chord('A2', 4.0))

# Bar 3: G Major (Power)
riff_stream.append(make_power_chord('G2', 4.0))

# Bar 4: G Minor (Full Triad for emotional impact)
# Note: Gm is NOT a power chord; we explicitly need the minor 3rd (Bb)
# to convey the "sadness" or "torn heart" (Jigar dareeda).
gm_chord = chord.Chord()
gm_chord.quarterLength = 4.0
riff_stream.append(gm_chord)
```

The shift from the open, hollow Power Chords to the dense, full Gm triad in the fourth bar serves as the emotional hook, programmatically enforcing the "Minor IV" theory outlined in the prompt.[1]

6.4 Movement IV: Synthesis (05:00 – End)

The final section is the "Synthesis." We need to bring the Sitar back, but force it to adapt to the Rock groove. This represents the poet's acceptance of reality—not abandoning love, but integrating it with the struggle.

The "Resolved" Sitar:

We will generate a Sitar melody that finally abandons the *Tivra Ma* (G#) and adopts the *Shuddha Ma* (G Natural) of the rock backing.

Python

```
def generate_synthesis_melody():
    synthesis_part = stream.Part()
    synthesis_part.insert(0, instrument.Sitar())

    # New Scale: Mixolydian/Major (No sharp 4)
    # The Sitar accepts the G Natural of the Revolution.
    safe_notes =
```

```
    # Generate a rhythmic, marching melody on Sitar
    # Locked to the 4/4 grid (no rubato, straight 8ths)
    for _ in range(8): # 8 bars
        n = note.Note(random.choice(safe_notes))
        n.quarterLength = 0.5 # Straight 8ths match the rock drums
        synthesis_part.append(n)

    return synthesis_part
```

7. Advanced Techniques: Microtonality and The "Meend"

One of the limitations of standard music21 MIDI export is the lack of fluid pitch bends (*Meend*), which are essential for a convincing Sitar performance. However, music21 allows us to define microtone objects attached to pitches, which can be interpreted by specialized playback systems or mapped to MIDI pitch wheel data.[3]

Implementing the Tivra Ma Tuning

Raag Yaman's *Tivra Ma* is often intoned slightly sharper than an equal-tempered F#/G# to add brilliance. We can explicitly set this cent deviation in our code.

Python

```
tivra_ma = note.Note('G#4')
# Sharpen by 10 cents to add "yearning" brilliance
tivra_ma.pitch.microtone = pitch.Microtone(10)
```

While standard MIDI players might ignore this, advanced synthesizers that support MIDI Tuning Standard (MTS) or pitch-bend data can interpret it. If exporting to MIDI for a DAW (like Ableton or Logic), we can write a function to convert these microtones into pitchBend messages on a separate channel to achieve the audible glissando effect.[16]

8. Putting It All Together: The Master Score

Finally, we assemble all parts into a master stream.Score. This function acts as the "Conductor," arranging the temporal sequence of the generated parts.

Python

```
def build_sahar_e_nau():
    score = stream.Score()
    score.metadata = metadata.Metadata(title='Sahar-e-Nau')

    # 1. Generate Parts via Helper Functions
    sitar = generate_alaap(duration_quarters=32)
    fracture = generate_fracture_texture()
    guitar_riff = stream.Part() # Populate with loop
    drums = create_keherwa_cycle() # Populate with loop

    # 2. Add Parts to Score with appropriate Offsets
    # Movement I (0-32 bars)
    score.insert(0, sitar)

    # Movement II (32-48 bars)
    score.insert(128, fracture) # Offset = 32 bars * 4 beats

    # Movement III (48+ bars)
    score.insert(192, guitar_riff)
    score.insert(192, drums)

    # 3. Export
    # score.write('midi', fp='sahar_e_nau.mid')
    # score.show() # Displays in MuseScore/Finale

    return score
```

9. Conclusion

By deconstructing *Sahar-e-Nau* into its musicological atoms—pitch classes, duration ratios, and timbral objects—we have created a Python framework that does more than play notes; it simulates a cultural dialogue. The music21 library allows us to codify the "Ma Paradox" and the "Minor IV" not as abstract concepts, but as executable classes. This computational rigor forces us to confront the precise nature of the fusion: exactly how many cents does the Sitar bend? Exactly what ratio defines the transition from feudal swing to revolutionary march?

The resulting code is a testament to the "Unified Theory of Faiz" proposed in the fusion document: distinct periods and styles (Classical and Revolutionary) can be woven into a

single, continuous symphony of human experience, mediated by the logic of code. The computational effort required to align these systems serves only to highlight the artistic genius required to fuse them in reality.

Works cited

1. Faiz Fusion: Classical to Revolutionary
2. The music21 Stream: A New Object Model for Representing ..., accessed January 27, 2026, https://quod.lib.umich.edu/i/icmc/bbp2372.2011.012/5/--music21-stream-a-new-object-model-for-representing-filtering?page=root;size=75;view=text
3. music21.pitch - Michael Scott Asato Cuthbert, accessed January 27, 2026, https://www.music21.org/music21docs/moduleReference/modulePitch.html
4. User's Guide, Chapter 19: Advanced Durations (Complex and Tuplets), accessed January 27, 2026, https://www.music21.org/music21docs/usersGuide/usersGuide_19_duration2.html
5. music21.stream.base - Michael Scott Asato Cuthbert, accessed January 27, 2026, https://www.music21.org/music21docs/moduleReference/moduleStreamBase.html
6. User's Guide, Chapter 6: Streams (II): Hierarchies, Recursion, and ..., accessed January 27, 2026, https://www.music21.org/music21docs/usersGuide/usersGuide_06_stream2.html
7. User's Guide, Chapter 58: Understanding Sites and Contexts, accessed January 27, 2026, https://www.music21.org/music21docs/usersGuide/usersGuide_58_sitesContext.html
8. Is 12/8 just 4/4 but with triplets instead of eighth notes? - Reddit, accessed January 27, 2026, https://www.reddit.com/r/musictheory/comments/c8ojqw/is_128_just_44_but_with_triplets_instead_of/
9. User's Guide: Chapter 14: Time Signatures and Beats — music21 ..., accessed January 27, 2026, https://www.music21.org/music21docs/usersGuide/usersGuide_14_timeSignatures.html
10. music21.instrument, accessed January 27, 2026, https://www.music21.org/music21docs/moduleReference/moduleInstrument.html
11. User's Guide, Chapter 4: Lists, Streams (I) and Output — music21 ..., accessed January 27, 2026, https://www.music21.org/music21docs/usersGuide/usersGuide_04_stream1.html
12. music21.articulations - Michael Scott Asato Cuthbert, accessed January 27, 2026, https://www.music21.org/music21docs/moduleReference/moduleArticulations.html
13. General MIDI (GM) Percussion Note Assignments ? B1(35), accessed January 27, 2026, https://midnightmusic.com/wp-content/uploads/2012/08/GMPercussion-and-Sib

elius-Drum-Map.pdf
14. music21.midi.percussion - Michael Scott Asato Cuthbert, accessed January 27, 2026, https://www.music21.org/music21docs/moduleReference/moduleMidiPercussion.html
15. Essential Free Guide To Tabla | PDF - Scribd, accessed January 27, 2026, https://www.scribd.com/doc/190008171/Essential-Free-Guide-to-Tabla
16. How to use pitch bend wheel without sounding like an idiot - Reddit, accessed January 27, 2026, https://www.reddit.com/r/synthesizers/comments/ldhzxd/how_to_use_pitch_bend_wheel_without_sounding_like/
17. MIDI Pitch Bend | Sound Examples - GitHub Pages, accessed January 27, 2026, https://tigoe.github.io/SoundExamples/midi-pitch-bend.html