B.TECH FINAL YEAR PROJECT

Music Composition Using Genetic Algorithm and Bresenham's Line Algorithm

Indian Institute of Technology (ISM) Dhanbad

Department of Computer Science and Engineering 2023-24

Project Guide

Prof Haider Banka
Associate Professor

Department of Computer Science and Engineering

Submitted By

Ishita Gupta
20JE0438

20je0438@cse.iitism.ac.in

AGENDA

INTRODUCTION

“Music is innately tied to the technology used to create it - as the tools evolve, so will the art.”

Music composition is a deeply expressive art form that transcends cultural boundaries, providing a unique channel for emotional communication. Creators engage with **melody**, **rhythm**, and **harmony** to craft compositions that resonate with listeners. Traditionally, this process has been driven by human intuition, creativity, and theoretical knowledge.

However, in the ever-evolving landscape of technology and artificial intelligence, there lies a fascinating **intersection** between creativity and algorithms. This intersection opens the door to innovative methods of music composition, blending the richness of human expression with the precision of computational models.

GENETIC ALGORITHM IN MUSIC COMPOSITION

Genetic algorithms, inspired by **natural selection,** are a potent force in musical innovation. Encoding musical elements into chromosomes and utilizing genetic operators like crossover and mutation allow for the iterative evolution of musical phrases. As a virtual composer, the algorithm explores vast musical spaces, generating compositions with unique qualities.

This presentation focuses on using genetic algorithms in music composition, emphasizing melody. Additionally, we'll explore the algorithm's augmentation to include harmony, unlocking new possibilities for rich and harmonically engaging pieces.

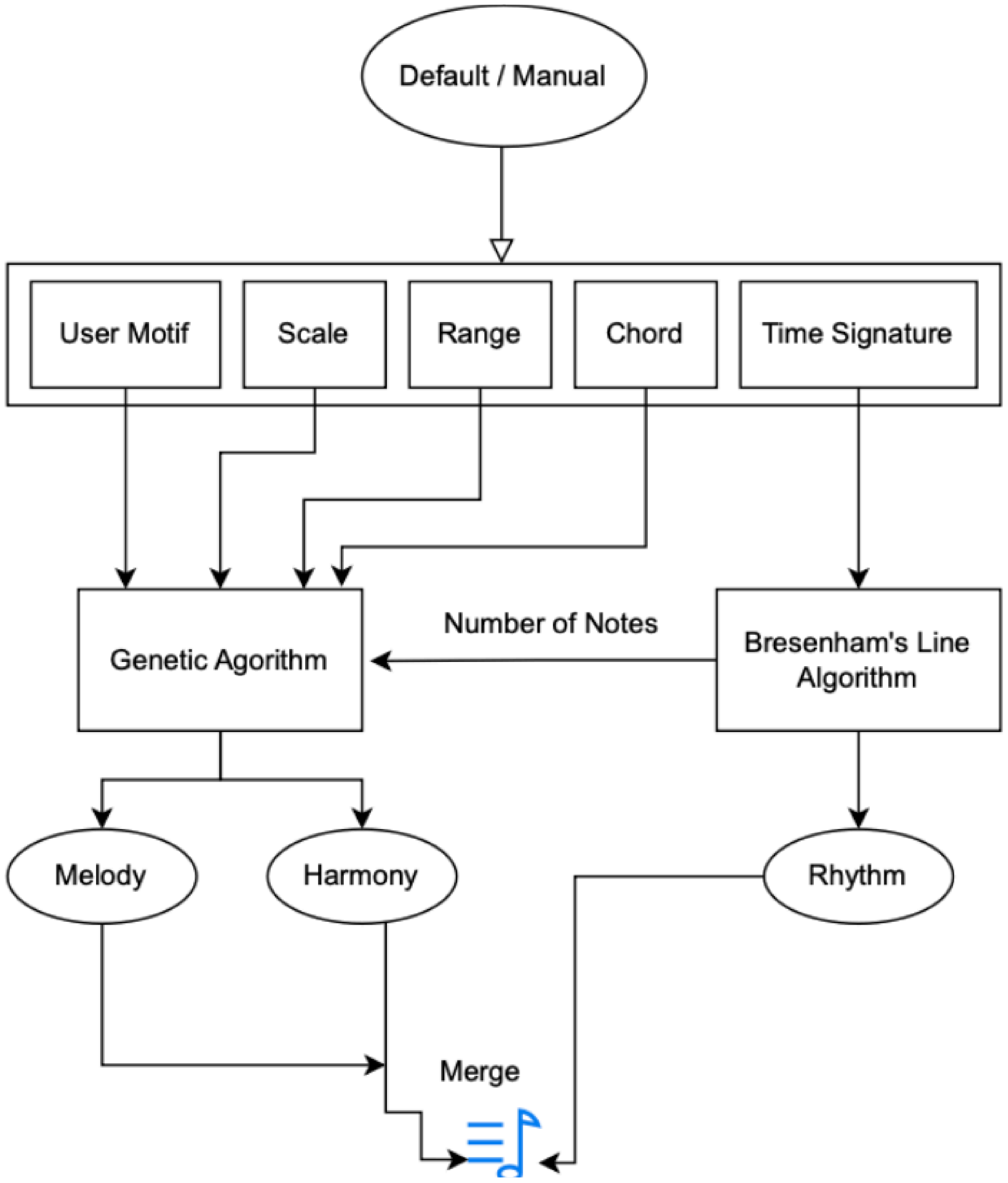
Through this innovative approach, we aim to push the boundaries of music composition at the intersection of art and technology.

BRESENHAM’S LINE ALGORITHM IN MUSIC COMPOSITION

Originally developed for computer graphics, Bresenham's algorithm efficiently draws straight lines on a raster display by using incremental error-based calculations.

In this presentation, we explore an intriguing adaptation of Bresenham's algorithm for rhythm generation in music.
Music often relies on rhythmic patterns, which can be represented as sequences of discrete time intervals. Bresenham's algorithm offers a unique approach to generating these patterns with efficiency and precision.

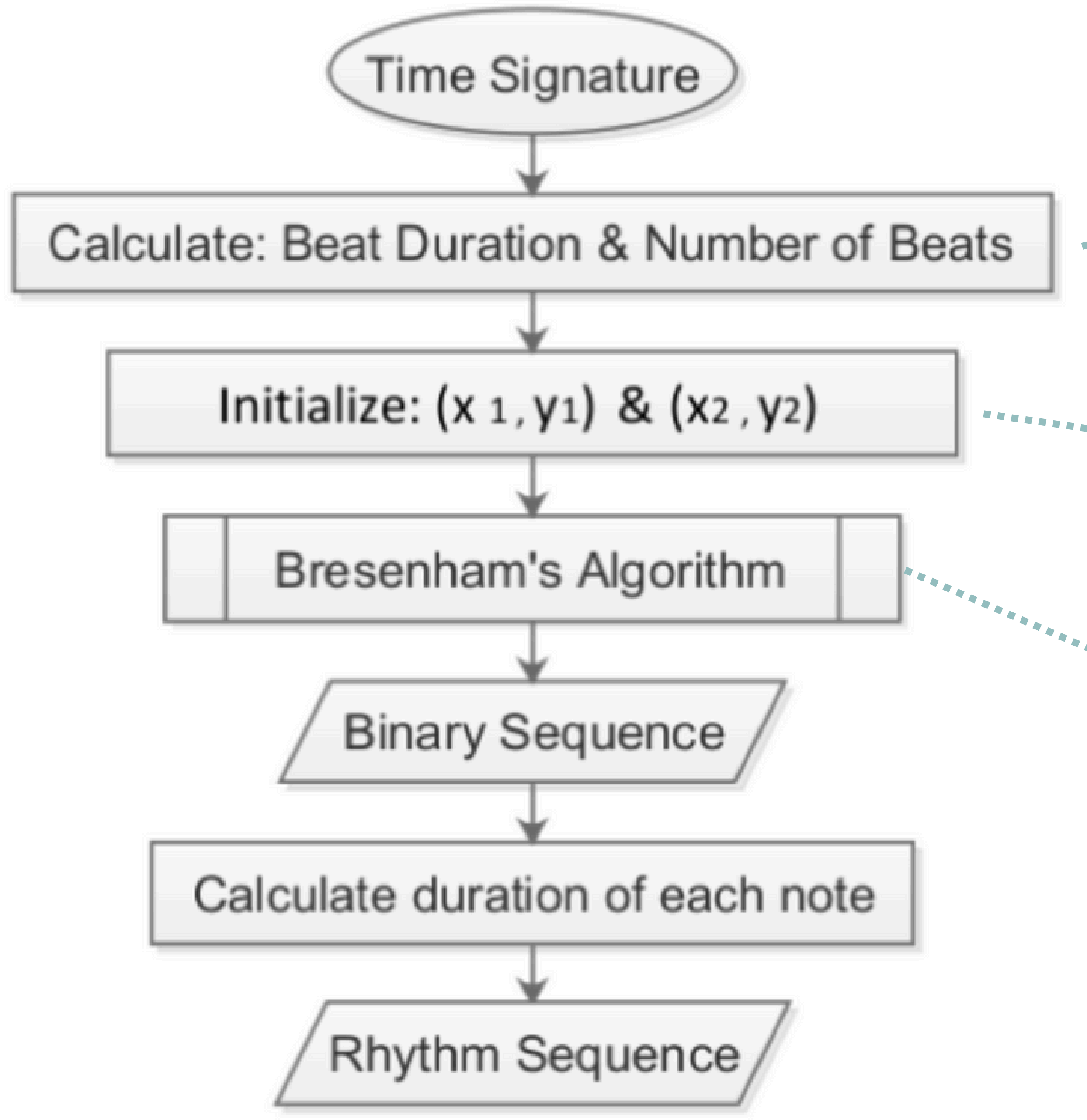
We will delve into how this algorithm, originally designed for visual rendering, can be repurposed to inspire and create rhythmic sequences, offering new possibilities for music composition and performance.



PROCESS FLOW CHART

RHYTHM GENERATION
(Bresenham's Line Algorithm)

Time Signature = Bt / V


Time Signature
Calculate: Beat Duration & Number of Beats
Initialize: (x_1, y_1) & (x_2, y_2)
Bresenham's Algorithm
Binary Sequence
Calculate duration of each note
Rhythm Sequence

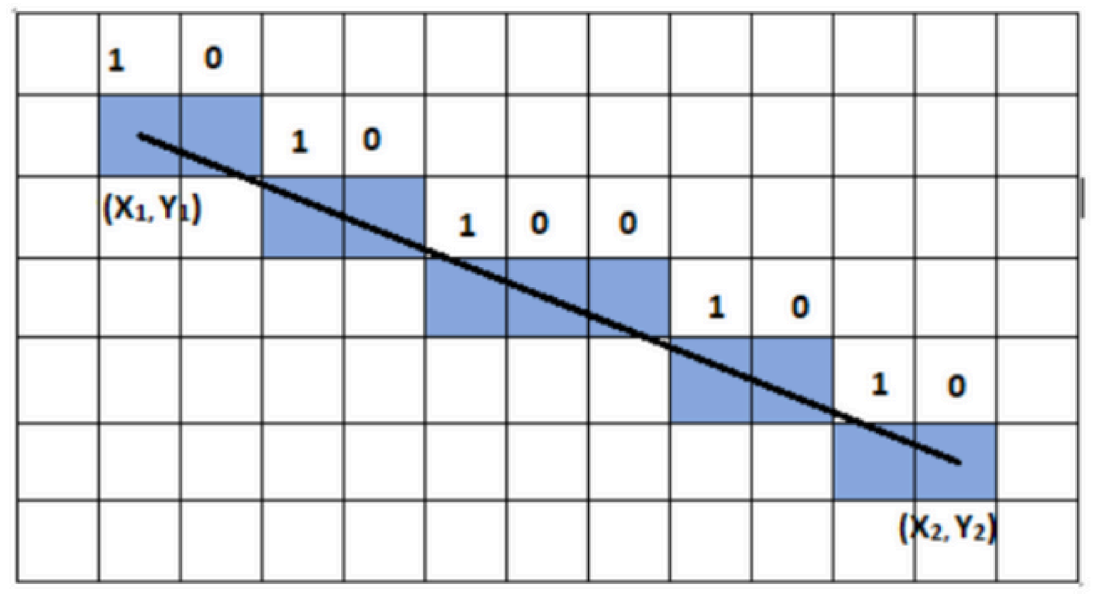
Beat Duration = (1/(2^n * V))
Number of Beats = 2^n(B* Bt)

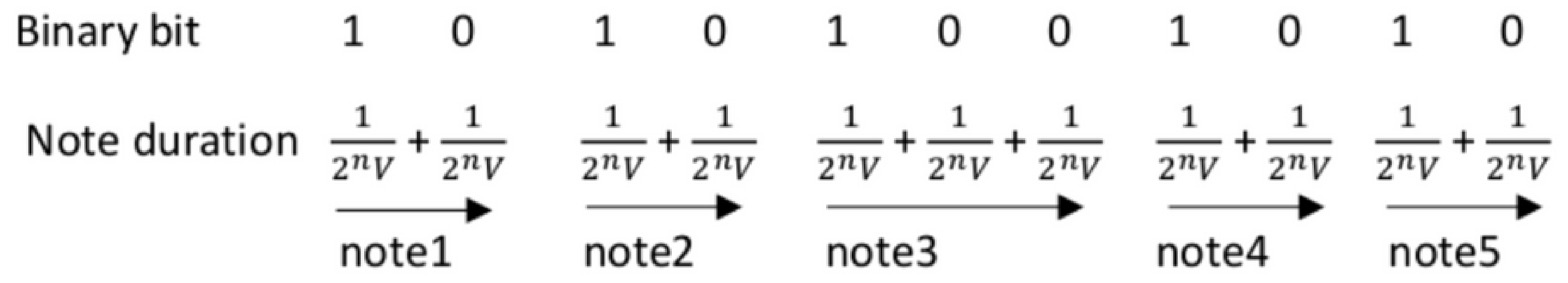
$$n = \left\{ x \mid x \in z,\ \frac{1}{16} \leq \frac{1}{2^x * V} \leq \frac{1}{2} \right\}$$

$$(x_1, y_1) = (0,0)$$
$$x_2 = 2^n (B * Bt)$$
$$y_2 = [0, x_2]$$

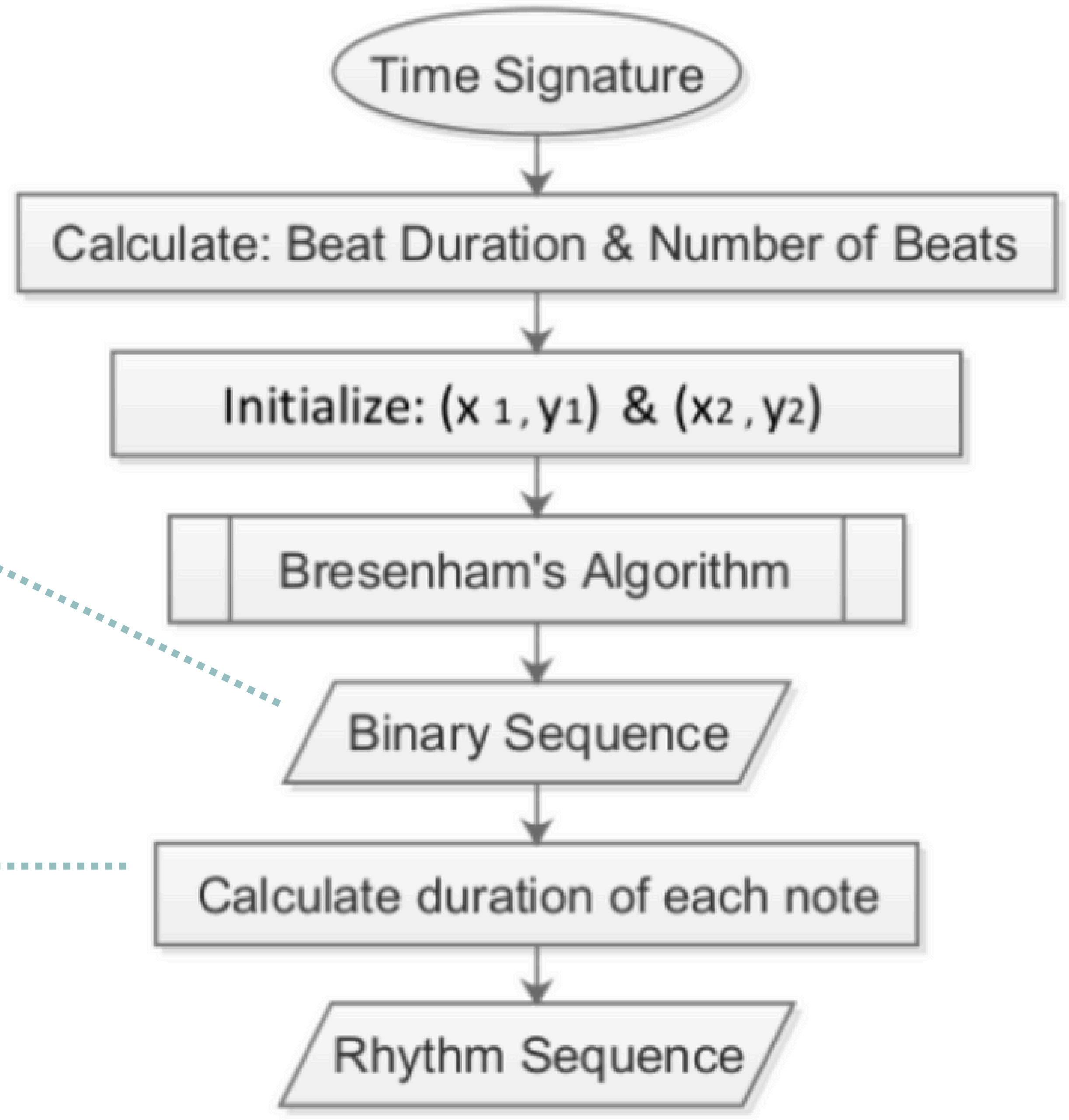
Bresenham's Line Algorithm for Rhythm
Input: Start point (x1, y1), End point (x2, y2)
Output: Binary Sequence (S)

RHYTHM GENERATION
(Bresenham’s Line Algorithm)





$\frac{1}{2^n V} + \frac{1}{2^n V}$ $\frac{1}{2^n V} + \frac{1}{2^n V}$ $\frac{1}{2^n V} + \frac{1}{2^n V} + \frac{1}{2^n V}$ $\frac{1}{2^n V} + \frac{1}{2^n V}$ $\frac{1}{2^n V} + \frac{1}{2^n V}$


Time Signature
Calculate: Beat Duration & Number of Beats
Initialize: (x1, y1) & (x2, y2)
Bresenham's Algorithm
Binary Sequence
Calculate duration of each note
Rhythm Sequence

The number of notes will serve as the length of the chromosome in melody generation.

GENETIC ALGORITHM STEPS

Step 1: Encoding Type

Define how potential solutions to a problem are represented as individuals or "chromosomes" in the algorithm. This encoding specifies how genetic information is structured and stored.

Step 2: Population Size

Determine the number of individuals that make up the population. A larger population can explore a more extensive solution space but may require more computational resources.

Step 3: Initial Population

Randomly generate the initial set of individuals in the population. These individuals represent possible solutions to the problem at hand and are typically created with random values.

Step 4: Parental Selection

(Roulette Wheel)

Evaluate the fitness of each individual in the population based on their performance in solving the problem. Use a selection method, such as roulette wheel selection, to choose individuals with higher fitness to be parents for the next generation.

GENETIC ALGORITHM STEPS

Step 5: Crossover

Apply crossover or recombination to the selected parents. This involves exchanging genetic information between two parents to create one or more offspring. Crossover introduces diversity and combines features from different parents.

Step 6: Mutation

Introduce small, random changes to the offspring's genetic information. Mutation adds diversity to the population and helps explore new regions of the solution space.

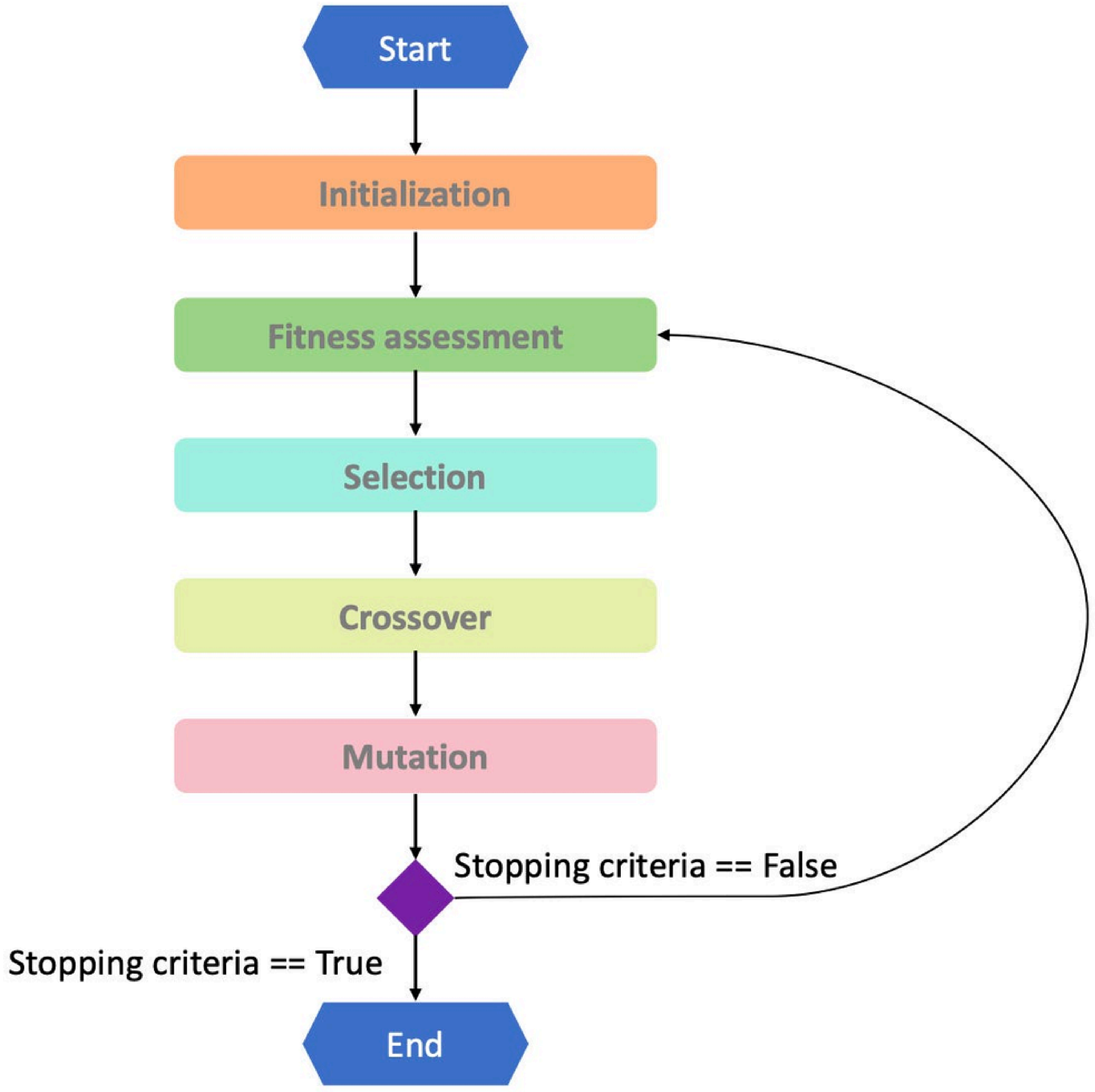
Step 7: Fitness Evaluation

Reassess the fitness of the newly created offspring in the population. Fitness is a measure of how well an individual solves the given problem and is used to guide the evolutionary process.

Step 8: Repetition

Repeat the entire process for a set number of iterations or until a stopping criterion is met. Over successive generations, the algorithm evolves and refines solutions, potentially converging towards optimal or near-optimal solutions.

STEPS INVOLVED



MELODY GENERATION

STEP 1 : Encoding Type

The encoding type for chromosome in this case is

<Note index, Octave index, Binary bit>

STEP 2 : Population Size

Potential size of **5** is chosen to represent potential music solutions.

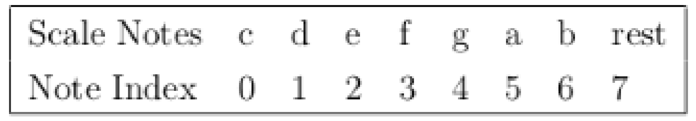
STEP 3 : Initial Population

<2, 0, 1> <4, 1, 1> <1, 2, 1> <5, 1, 0> <3, 0, 1>

(Randomly selected)

Note Index (NI) represents the note of the scale. Notes along with rest are encoded in decimal numbers. (Table 3.2)

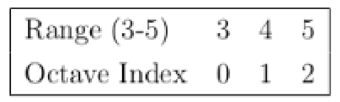
Table 3.2: Example of note index mapping



Scale Notes	c	d	e	f	g	a	b	rest
Note Index	0	1	2	3	4	5	6	7

Octave Index (OI) represents the octave of the current note. These octaves are also encoded in decimal number based on the user given range. (Table 3.3)

Table 3.3: Example of octave index mapping



Range (3-5)	3	4	5
Octave Index	0	1	2

Binary Bit '1' is appended to the notes belonging to user's motif while bit '0' is appended to the randomly generated notes. (Figure 3.5)

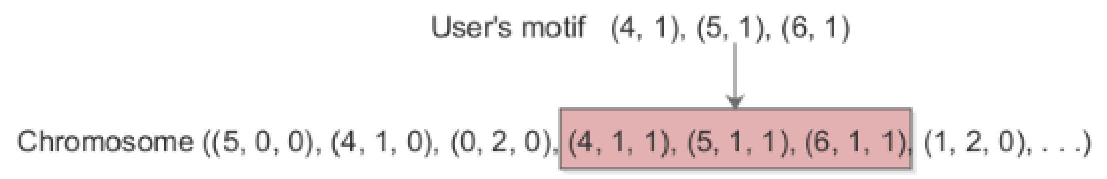

User's motif (4, 1), (5, 1), (6, 1)
Chromosome ((5, 0, 0), (4, 1, 0), (0, 2, 0), (4, 1, 1), (5, 1, 1), (6, 1, 1), (1, 2, 0), . . .)

Figure 3.5: Example of chromosome with user's motif.

MELODY GENERATION

STEP 4 : Fitness Function

Fitness = w1*ON + w2*TF + w3*NJ + w4*RN = 1*2 + 1*2 + 1*1 + (-1)*0 = **5**

- **Octave Note Fitness (ON):** Keeping the notes within an octave makes a melody singable without strain. This function counts the number of notes found in the same octave within a chromosome. Then returns the largest number of notes found in an octave.

- **Triad Fitness (TF):** A melody that remains on the same pitch can easily get boring. Good tunes have rising and falling melodies. As the melody proceeds, the pitches can gradually or rapidly go up or down. This fitness function looks at every set of three adjacent notes and awards a point if they either all rise or fall. As we can not directly compare the notes of different octaves, so we calculate the absolute value of the note using Eq. 3.2. Then we use Eq. 3.3 to find the triad fitness of the given chromosome. A constant value 2 is added in the Eq. 3.3 so that the maximum triad fitness is equal to the length of chromosome.

$$TF = 2 + \sum_{i=1}^{N} f_1(Ab_i, Ab_{i+1}, Ab_{i+2}) \tag{3.3}$$

$$f_1(Ab_i, Ab_{i+1}, Ab_{i+2}) = \begin{cases} 1, & Ab_i < Ab_{i+1} < Ab_{i+2} \\ 1, & Ab_i > Ab_{i+1} > Ab_{i+2} \\ 0, & otherwise \end{cases}$$

where Ab_i is the absolute value of i^{th} note in the chromosome, OI is octave index, NI is note index and S is number of notes in the scale.

- **No Jump Fitness (NJ):** Singable tunes do not contain large intervals. The function defined in Eq. 3.4 awards a point for every jump that is within a perfect fourth. A constant value 1 is added in the Eq. 3.4 so that the maximum fitness is equal to the length of chromosome.

$$NJ = 1 + \sum_{i=1}^{N} f_2(Ab_i, Ab_{i+1}) \tag{3.4}$$

$$f_1(Ab_i, Ab_{i+1}) = \begin{cases} 1, & |Ab_i - Ab_{i+1}| < MaxJump \\ 0, & otherwise \end{cases}$$

- **Repeat Note Fitness (RN):** Too much repetition of a note in consecutive manner does not sound good. This fitness function penalizes the genome having consecutive note repetition. Penalty is equal to the maximum consecutive occurrence of a note within a chromosome.

HARMONY GENERATION

STEP 1 : Encoding Type

The encoding type for chromosome in this case is

<Note index, Octave index, Binary bit, Chord Index>

The "Chord Index" is a numerical value assigned to each note in the chromosome. Each chord index corresponds to a specific chord or harmonic context.

0: No specific harmony
1: Harmony 1 (e.g., major chord)
2: Harmony 2 (e.g., minor chord)
3: Harmony 3 (e.g., diminished chord)

STEP 2 : Population Size

Potential size of **5** is chosen to represent potential music solutions.

STEP 3 : Initial Population

<2, 0, 1, 0> <4, 1, 1, 2> <1, 2, 1, 3> <5, 1, 0, 2> <3, 0, 1, 1>

(Randomly selected)

HARMONY GENERATION

STEP 4 : Fitness Function (modified)

Fitness = w1*ON + w2*TF + w3*NJ + w4*RN + w5*HF + w6*CP

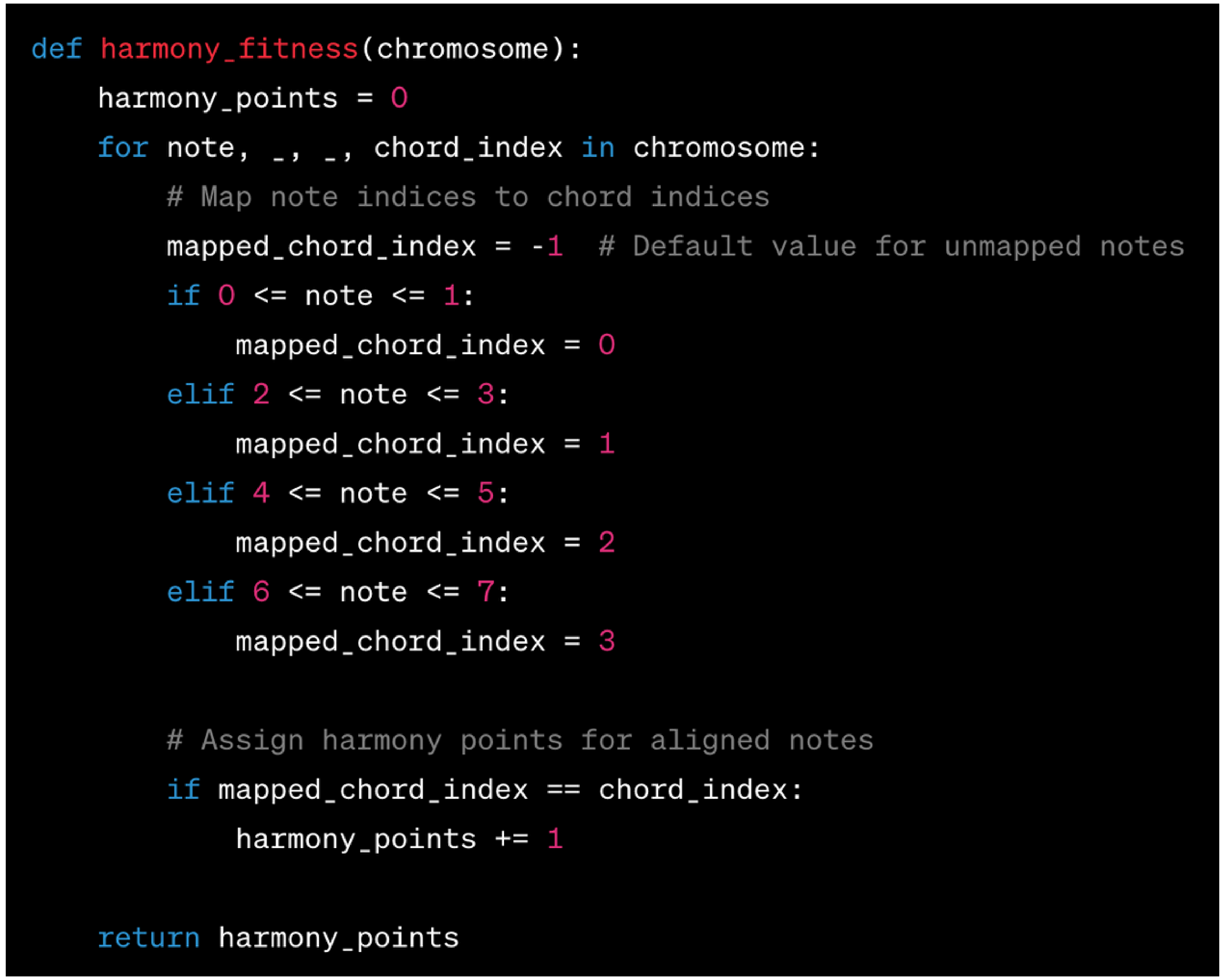
= 1*2 + 1*2 + 1*1 + (-1)*0 + 1*3 + 1*3 = **11**

HARMONY FITNESS

The harmony_fitness function aims to measure how well the melody notes align with specified chord indices. In the provided implementation, we've divided the range of note indices (0 to 7) into four regions, each corresponding to a specific chord index (0 to 3). The function awards points for each melody note that aligns with its corresponding chord index.

For note indices 0 and 1, the mapped chord index is 0.
For note indices 2 and 3, the mapped chord index is 1.
For note indices 4 and 5, the mapped chord index is 2.
For note indices 6 and 7, the mapped chord index is 3.

If the chord index of a melody note matches its mapped chord index, the function increments the harmony_points. This way, the function rewards consonant relationships between melody notes and the specified chords.



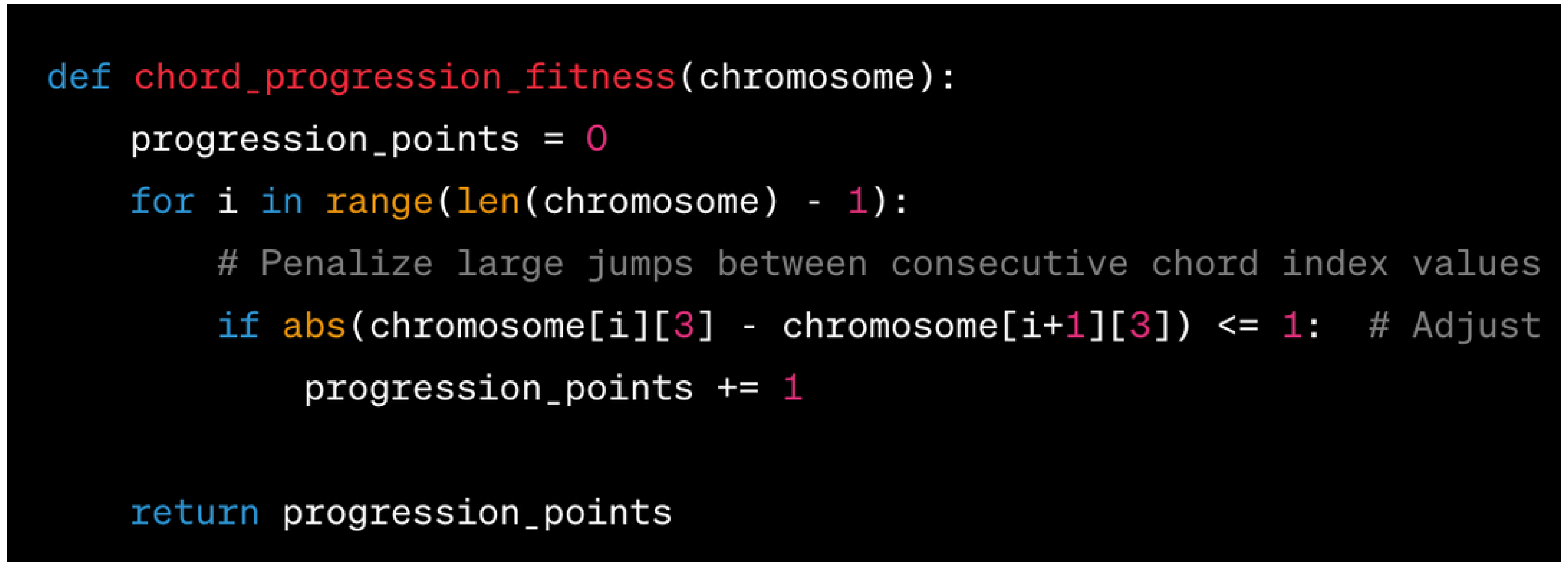
```python
def harmony_fitness(chromosome):
    harmony_points = 0
    for note, _, _, chord_index in chromosome:
        # Map note indices to chord indices
        mapped_chord_index = -1  # Default value for unmapped notes
        if 0 <= note <= 1:
            mapped_chord_index = 0
        elif 2 <= note <= 3:
            mapped_chord_index = 1
        elif 4 <= note <= 5:
            mapped_chord_index = 2
        elif 6 <= note <= 7:
            mapped_chord_index = 3

        # Assign harmony points for aligned notes
        if mapped_chord_index == chord_index:
            harmony_points += 1

    return harmony_points
```

HARMONY GENERATION

STEP 4 : Fitness Function

Fitness = w1*ON + w2*TF + w3*NJ + w4*RN + w5*HF + w6*CP

= 1*2 + 1*2 + 1*1 + (-1)*0 + 1*3 + 1*3 = **11**



```python
def chord_progression_fitness(chromosome):
    progression_points = 0
    for i in range(len(chromosome) - 1):
        # Penalize large jumps between consecutive chord index values
        if abs(chromosome[i][3] - chromosome[i+1][3]) <= 1:  # Adjust
            progression_points += 1

    return progression_points
```

CHORD PROGRESSION

The chord_progression_fitness function evaluates the quality of chord progressions in the melody. It awards points for smooth transitions between consecutive chords and penalizes large jumps between chord indices. The provided implementation checks for the absolute difference between consecutive chord indices and awards points for smaller differences.

The specific condition abs(chromosome[i][3] - chromosome[i+1][3]) <= 1 checks if the chord progression has a difference of 1 or less between consecutive chords. You may adjust this threshold based on your preferences. Smaller differences suggest smoother progressions, while larger differences may indicate less harmonically connected chord changes.

FIRST ITERATION OF GENETIC ALGORITHM

Based on the modified parameters after harmony inclusion, first iteration of Genetic Algorithm looks like the following :

STEP 1 : Encoding Type = <Note index, Octave index, Binary bit, Chord index>

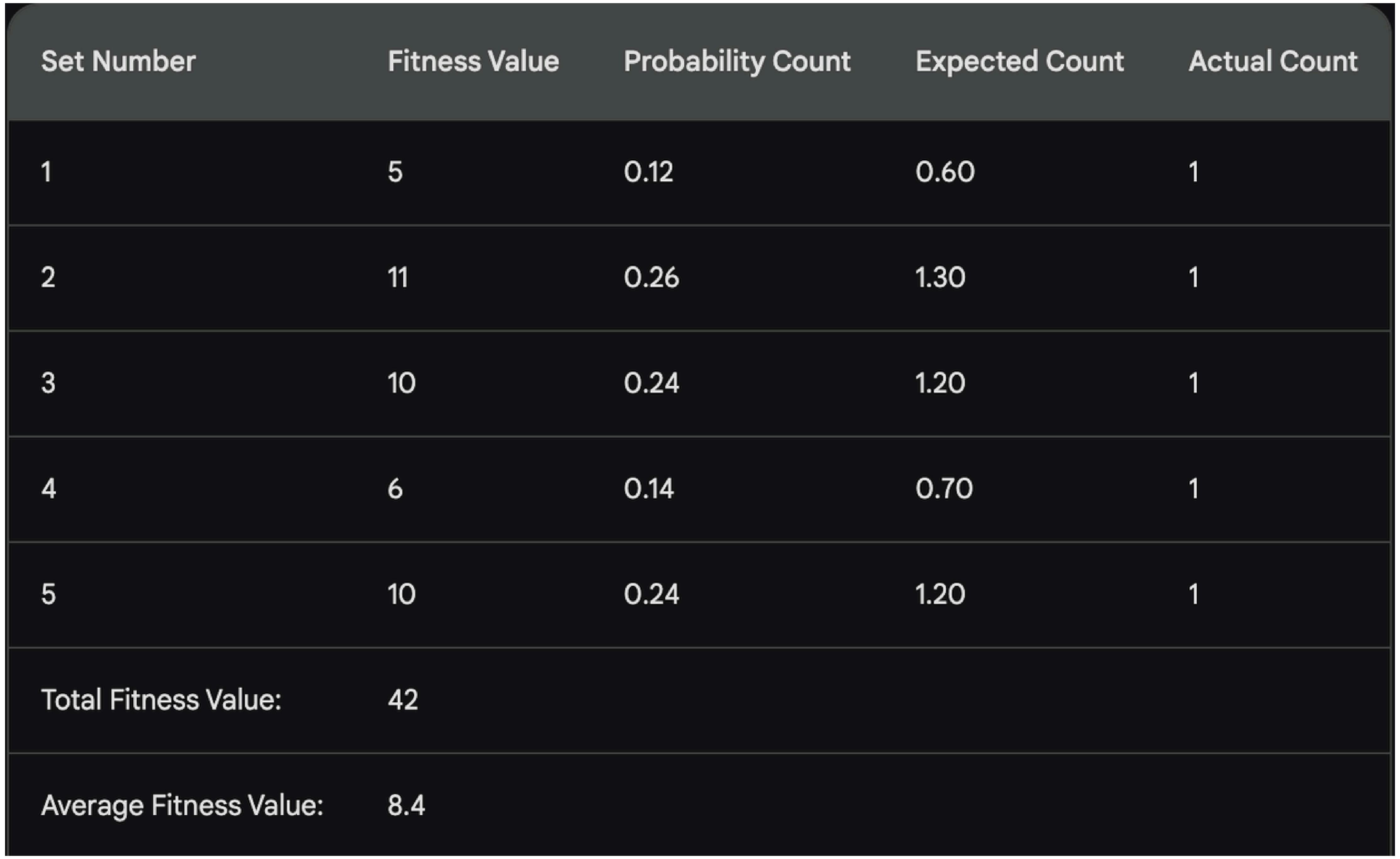
STEP 2 : Population Size = 5

STEP 3 : Initial Population (randomly chosen)

Set 1 --> 1: (3, 1, 0, 2) 2: (6, 2, 1, 0) 3: (1, 0, 1, 3) 4: (5, 1, 0, 1) 5: (2, 2, 0, 3)
Set 2 --> 1: (2, 0, 1, 3) 2: (4, 1, 0, 2) 3: (0, 2, 1, 1) 4: (7, 0, 0, 0) 5: (3, 2, 1, 1)
Set 3 --> 1: (5, 1, 1, 0) 2: (1, 0, 0, 2) 3: (3, 2, 1, 1) 4: (6, 1, 0, 3) 5: (4, 0, 1, 3)
Set 4 --> 1: (0, 2, 0, 3) 2: (4, 0, 1, 1) 3: (7, 1, 0, 2) 4: (2, 2, 1, 0) 5: (5, 0, 0, 3)
Set 5 --> 1: (3, 0, 0, 1) 2: (6, 1, 1, 3) 3: (1, 2, 0, 0) 4: (5, 0, 1, 2) 5: (2, 1, 1, 3)

FIRST ITERATION OF GENETIC ALGORITHM

STEP 4 : Parent Selection (Roulette Wheel Selection Method)



Set Number	Fitness Value	Probability Count	Expected Count	Actual Count
1	5	0.12	0.60	1
2	11	0.26	1.30	1
3	10	0.24	1.20	1
4	6	0.14	0.70	1
5	10	0.24	1.20	1
Total Fitness Value:	42			
Average Fitness Value:	8.4			

Probability Count(x) =
Fitness Value(x) / Total

Expected Count(x) =
Fitness Value(x) / Average

Actual Count(x) =
approx(Expected Count(x))

Based on the following result, **set 2** and **set 3** are selected as parents because of high fitness values.

FIRST ITERATION OF GENETIC ALGORITHM

STEP 5 : Crossover (Single Point Crossover at point **3**)

Parent 1 = Set 2: 1: (2, 0, 1, 3) 2: (4, 1, 0, 2) | 3: (0, 2, 1, 1) 4: (7, 0, 0, 0) 5: (3, 2, 1, 1)
Parent 2 = Set 3:1: (5, 1, 1, 0) 2: (1, 0, 0, 2) | 3: (3, 2, 1, 1) 4: (6, 1, 0, 3) 5: (4, 0, 1, 3)

Crossover 1 = 1: (2, 0, 1, 3) 2: (4, 1, 0, 2) 3: (3, 2, 1, 1) 4: (6, 1, 0, 3) 5: (4, 0, 1, 3)
Crossover 2 = 1: (5, 1, 1, 0) 2: (1, 0, 0, 2) 3: (0, 2, 1, 1) 4: (7, 0, 0, 0) 5: (3, 2, 1, 1)

STEP 6 : Mutation

Harmony Mutation : 1st chromosome , Octave Mutation : 2nd chromosome and
Note Mutation + Harmony Mutation : 4th chromosome

Offspring 1: 1: (2, 0, 1, 1) 2: (4, 0, 0, 2) 3: (3, 2, 1, 1) 4: (3, 1, 0, 1) 5: (4, 0, 1, 3)
Offspring 2: 1: (5, 1, 1, 2) 2: (1, 2, 0, 2) 3: (0, 2, 1, 1) 4: (5, 0, 0, 2) 5: (3, 2, 1, 1)

STEP 7 : Fitness Evaluation

Fitness for offspring 1 = 17
Fitness for offspring 2 = 15

STEP 8 : Repetition

CODE IMPLEMENTATION

bresenhams_line_algo.py

The “calculate_beat_parameters” function calculates the total number of beats and the note duration based on the given time signature (e.g., 4/4).

The “bresenhams_line_algorithm” function implements Bresenham's line algorithm to generate a binary sequence representing a line between two points in 2D space. Each step along the line corresponds to a note, and the binary sequence indicates whether a note should be played or not.

genetic_algo_fitness.py

“generate_initial_population” function generates an initial population of motifs. Each motif is represented as a list of tuples, where each tuple contains information about a note (e.g., note index, octave index, binary bit, and chord index).

“calculate_overall_fitness” combines the individual fitness measures(octave_note_fitness, triad_fitness, no_jump_fitness, repeat_note_fitness, harmony_fitness, chord_progression_fitness) using weighted sums to calculate the overall fitness.

CODE IMPLEMENTATION

selection.py

“Calculate Normalized Probabilities”: Normalize the fitness values of the population to convert them into probabilities. This step ensures that the probabilities sum up to 1.
“Sort Population by Probabilities”: Sort the population based on their normalized probabilities in descending order.
“Select Parents”: Select the top two parents from the sorted population. The selection is based on their probabilities, with fitter individuals having higher probabilities of being selected.
“Return Selected Parents”: Return the selected parents along with their corresponding fitness values.

crossover_and_mutation.py

“crossover”: This function performs crossover between two parents. It utilizes the calculated melodic intervals to determine the crossover point that minimizes the melodic interval. Then, it creates two children by combining parts of the parents before and after the crossover point.
“note_based_mutation”: This function implements note-based mutation in a chromosome. It iterates through each note in the chromosome and checks if the difference between neighboring notes exceeds a threshold (4 in this case). If the difference is greater than the threshold, it replaces the note with a new note within a certain range around the neighboring note's index.

CODE IMPLEMENTATION

mapping.py

“map_chromosome_to_notes_duration”: This function takes a chromosome and a binary sequence as input. It identifies sequences of 1s in the binary sequence, which represent notes to be played, and calculates their durations based on the provided single note duration. It returns a list of note durations.
“convert_chromosome_to_note_number”: This function maps each gene (tuple) in the chromosome to a MIDI note number based on a predefined mapping. It iterates through the chromosome, looks up each gene in the mapping dictionary, and appends the corresponding MIDI note number to a list. Finally, it returns the list of MIDI note numbers.

midi_generation.py

“notes_to_midi”: This function takes a list of notes (each represented by a tuple of MIDI note number and duration) as input, along with optional parameters for the output file name and tempo. It creates a new MIDI file and track, adds a tempo meta message to the track, and calculates ticks per beat based on the MIDI file's default settings.
“Conversion to MIDI Messages”: For each note in the input list, it calculates the corresponding ticks based on the note's duration and tempo. Then, it adds note-on and note-off messages to the MIDI track with appropriate timing information.
“Saving the MIDI File”: Finally, it saves the MIDI file with the specified output file name.

CODE IMPLEMENTATION

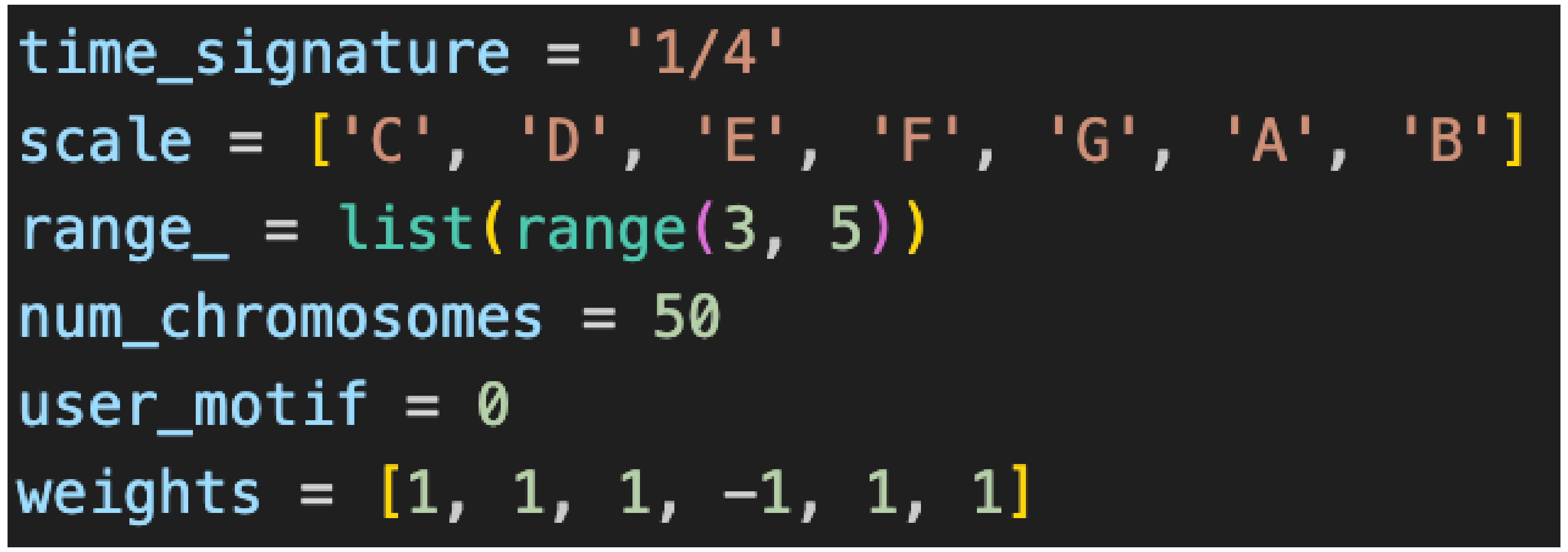
main.py

Step 1: Calculate beat parameters based on the time signature.
Step 2: Initialize endpoints for Bresenham's line algorithm.
Step 3: Perform Bresenham's line algorithm to generate a binary sequence representing note events.
Step 4: Initialize a population of motifs for the genetic algorithm.

Step 5: Calculate the fitness of each motif in the population.
Step 6: Perform roulette wheel selection to select top 2 parents based on their fitness.
Step 7: Apply crossover and mutation to generate child motifs from the selected parents.
Step 8: Map the child motifs into (note_number, note_duration) format.
Step 9: Generate a MIDI file from the mapped motifs.

RESULTS & DISCUSSIONS

INPUT



```
time_signature = '1/4'
scale = ['C', 'D', 'E', 'F', 'G', 'A', 'B']
range_ = list(range(3, 5))
num_chromosomes = 50
user_motif = 0
weights = [1, 1, 1, -1, 1, 1]
```

OUTPUT

output_1.mp3

output_2.mp3

output_3.mp3

FUTURE SCOPE

1. Inclusion of User Motif

- *Objective: Users can add some music element that they wish to be a part of the generated music.*
- *Benefits: Makes the generated music more personalised as it includes some user defined element.*

2. Fine-Tuning Algorithm Parameters

- *Objective*: Continuously refine and optimize algorithm parameters to encourage more diverse and creative musical outputs.
- *Benefits*: Allows for a nuanced control over the algorithm's behavior, adapting to various musical styles.

3. Exploration into Real-time Composition

- *Objective*: Investigate the feasibility of real-time genetic algorithm-driven music composition.
- *Benefits*: Enables dynamic and interactive music creation, opening possibilities for live performances and installations.

4. Sole Dependence of Genetic Algorithm

- *Objective: Generating rhythm also through Genetic Algorithm.*
- *Benefits: No need for extra algorithm. Makes the model simpler.*

CONCLUSION

1. Harmonious Integration

   The amalgamation of genetic algorithms with harmony components has yielded compositions that are more melodically rich and harmonically engaging.

2. Evolutionary Progress

   Through successive iterations, the algorithm demonstrated an inherent capacity to evolve and refine musical output, showcasing its adaptability.

3. Improved Fitness Landscape

   The modification of algorithm parameters positively impacted the overall fitness landscape, indicating a successful enhancement of musical quality.

The exploration into genetic algorithms for music composition has showcased a promising avenue for creative expression, blending computational efficiency with artistic ingenuity.

Thank You for listening!

Ishita Gupta