MUSIC COMPOSITION USING GENETIC ALGORITHM AND BRESENHAM'S LINE ALGORITHM

by

Ishita Gupta

20JE0438

Submitted to Indian Institute of Technology (ISM) Dhanbad in fulfilment of the requirements for the award of the degree of

Bachelor of Technology in
Computer Science and Engineering

Under the guidance of
Prof. Haider Banka

Department of Computer Science and Engineering
Indian Institute of Technology (Indian School of Mines), Dhanbad

MAY 08, 2024

कम्पयूटर विज्ञान एवं अभियांत्रिकी विभाग
DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
भारतीय प्रौद्योगिकी संस्थान (भारतीय खनि विद्यापीठ), धनबाद
धनबाद-826004, झारखण्ड, भारत
INDIAN INSTITUTE OF TECHNOLOGY (INDIAN SCHOOL OF MINES), DHANBAD
DHANBAD-826004, JHARKHAND, INDIA

CERTIFICATE

This is to certify that the thesis titled "MUSIC COMPOSITION USING GENETIC ALGORITHM AND BRESENHAM'S LINE ALGORITHM" being submitted by Ms. Ishita Gupta (Admission no: 20JE0438) for the award of the degree of Bachelor of Technology in the Department of Computer Science and Engineering of Indian Institute of Technology (Indian School of Mines), Dhanbad is a record of bonafide research work carried out by them under my supervision. In my opinion, the thesis is worthy of consideration for the award of the degree of Bachelor of Technology in accordance with the regulations of the institute. The results presented in the thesis have not been submitted to any other university or institute for the award of any degree.

[Signature of Guide]
Dr. Haider Banka
Associate Professor
Department of Computer Science and Engineering,
Indian Institute of Technology (ISM) Dhanbad-826004, Jharkhand, INDIA

Date: 08/05/2024

कम्पयूटर विज्ञान एवं अभियांत्रिकी विभाग
DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
भारतीय प्रौद्योगिकी संस्थान (भारतीय खनि विद्यापीठ), धनबाद
धनबाद-826004, झारखण्ड, भारत
INDIAN INSTITUTE OF TECHNOLOGY (INDIAN SCHOOL OF MINES), DHANBAD
DHANBAD-826004, JHARKHAND, INDIA

CERTIFICATE FOR CLASSIFIED DATA

This is to certify that the Dissertation entitled "MUSIC COMPOSITION USING GENETIC ALGORITHM AND BRESENHAM'S LINE ALGORITHM" being submitted to the Indian Institute of Technology (Indian School of Mines), Dhanbad by Ms. Ishita Gupta for the award of Bachelor Degree in Computer Science and Engineering does not contain any classified information. This work is original and yet not been submitted to any institution or university for the award of any degree.

Signature of Guide	Signature of Student
Dr. Haider Banka	Ishita Gupta

कम्पयूटर विज्ञान एवं अभियांत्रिकी विभाग
DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
भारतीय प्रौद्योगिकी संस्थान (भारतीय खनि विद्यापीठ), धनबाद
धनबाद-826004, झारखण्ड, भारत
INDIAN INSTITUTE OF TECHNOLOGY (INDIAN SCHOOL OF MINES), DHANBAD
DHANBAD-826004, JHARKHAND, INDIA

CERTIFICATE REGARDING ENGLISH CHECKING

This is to certify that the Dissertation entitled "MUSIC COMPOSITION USING GENETIC ALGORITHM AND BRESENHAM'S LINE ALGORITHM" being submitted to the Indian Institute of Technology (Indian School of Mines), Dhanbad by Ms. Ishita Gupta for the award of Bachelor Degree in Computer Science and Engineering has been thoroughly checked for quality of English and logical sequencing of topics. It is hereby certified that the standard of English is good and that grammar and typos have been thoroughly checked.

Signature of Guide	Signature of Student
Dr. Haider Banka	Ishita Gupta

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to everyone who has supported me throughout my academic journey and the completion of this thesis.

First and foremost, I would like to thank my thesis supervisor Dr. Haider Banka for his guidance, expertise, and patience throughout the entire process. I am grateful for his valuable insights, constructive feedback, and unwavering support, which have been instrumental in shaping this research and helping me navigate the challenges along the way.

I am also deeply grateful to the faculty members of the Computer Science and Engineering department for their encouragement and support, particularly those who have provided me with their valuable time and resources for this project. Their insights and feedback have been invaluable in shaping our thinking and improving the quality of our work.

I would also like to express my sincere appreciation to the dedicated Research Scholars of the Department of Computer Science & Engineering at IIT (ISM) Dhanbad. Their generous allocation of time and valuable mentorship has been instrumental in shaping my research journey, and for this, I am deeply thankful.

Furthermore, I extend my gratitude to my fellow batchmates and my parents for their unwavering support and well wishes.

Ms. Ishita Gupta

Date: 08/05/2024

TABLE OF CONTENTS

INTRODUCTION

In the rich tapestry of music composition, the intricate interplay between art and technology is unmistakable. As the tools employed in crafting music undergo continual evolution, so too does the very essence of the art form. Music, a universal language transcending cultural confines, has long been shaped by human intuition, creativity, and theoretical expertise. Yet, within the dynamic realm of technology and artificial intelligence, a captivating convergence emerges between human ingenuity and algorithmic precision.

This fusion heralds a new era in music composition, where creators leverage computational models to enhance their artistic endeavours. The delicate interplay of melody, rhythm, and harmony finds a contemporary companion in algorithms, offering novel avenues to amplify the expressive potential of musical creations. This symbiotic relationship not only broadens the horizons of creativity but also challenges traditional boundaries within the sonic realm.

By marrying human emotional depth with the analytical prowess of artificial intelligence, a harmonious balance is achieved, giving rise to compositions that resonate on both intellectual and emotional levels. The evolving technological landscape serves as a catalyst for unprecedented sonic explorations, empowering artists to push the boundaries of conventional musical norms. Within this ongoing dialogue between art and technology, music becomes a dynamic reflection of the zeitgeist, a testament to the adaptability and resilience of this timeless form of expression.

GENETIC ALGORITHM IN MUSIC COMPOSITION

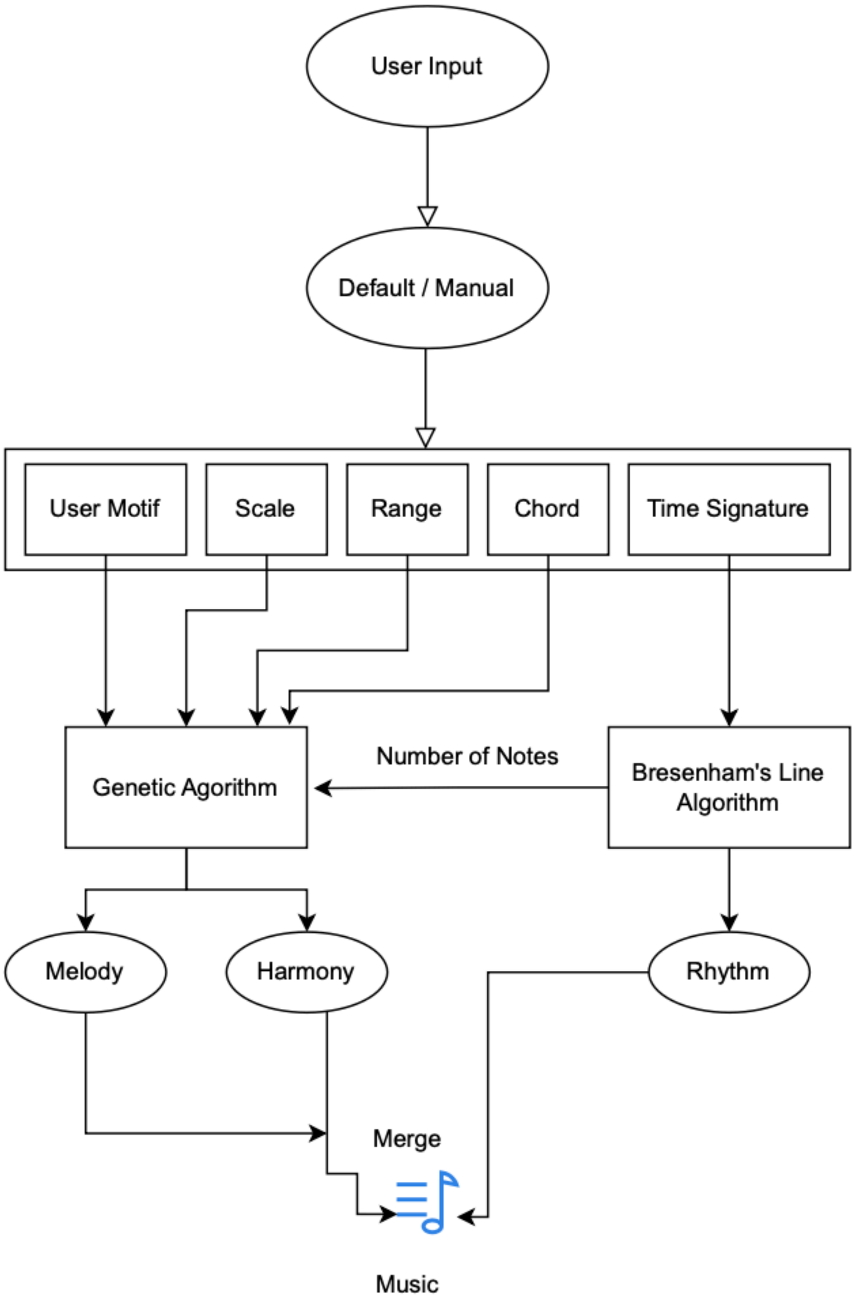
Within music composition, the integration of genetic algorithms introduces a novel and sophisticated dimension to the creative process. These algorithms, inspired by natural selection and evolution, offer a fresh computational approach to generating musical compositions. In this framework, musical elements are treated as genetic material subject to mutation, crossover, and selection, akin to the evolutionary mechanisms in biological systems.

This method empowers composers to navigate vast musical landscapes, using computation to refine and evolve compositions iteratively. Across successive generations of musical "genes," the algorithm hones its output based on predefined fitness criteria, allowing only the most musically compelling sequences to survive and propagate.

The collaboration between human artistic intent and algorithmic exploration is evident as composers guide the evolutionary process, influencing the algorithm's decisions. This collaborative interplay results in compositions that possess a unique blend of computational precision and human-inspired creativity.

The application of genetic algorithms in music composition not only diversifies the creative toolkit but also challenges traditional notions of authorship and creativity. This project delves into the potential of genetic algorithms as a groundbreaking tool for musical innovation, exploring the intricate interplay between code and composition to redefine the boundaries of musical expression in the digital age.

PROCESS FLOW CHART



RHYTHM GENERATION USING BRESENHAM'S LINE ALGORITHM

Rhythm: Rhythm is an essential and inescapable element of music. It refers to the organisation of time in music, and it is created through the use of regular and irregular patterns of beats and accents. Rhythm is an important element of music, and it can be used to create a sense of movement, momentum, and groove.

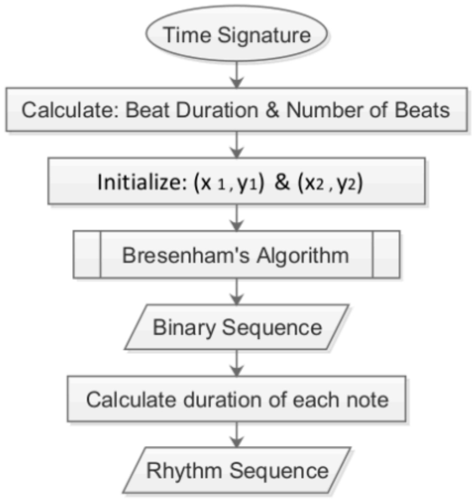

Time Signature
Calculate: Beat Duration & Number of Beats
Initialize: (x_1, y_1) & (x_2, y_2)
Bresenham's Algorithm
Binary Sequence
Calculate duration of each note
Rhythm Sequence

Figure 3.2: Overview of rhythm generation process.

The time signature given by the user is used to calculate beat duration and total number of beats in the rhythm. The rhythm pattern is generated for a single phrase having eight bars. Figure 3.2 illustrates the steps involved in the process. To understand the process of rhythm generation, let us consider the case with 8 bars and 4/4 time signature. Now, we can obtain the following values.

Beats per bar (Bt) = Top number of Time Signature = 4
Beat value (V) = Bottom number of Time Signature = 4
Number of bars (B) = 8

We calculate the total length of rhythm by taking the beat duration w.r.t. beat value. In this example the beat value is 4 so the duration of beat is 1/4.

$$\begin{aligned}\text{Length of rhythm} &= B * Bt * (1/V)\\ &= 8 * 4 * (¼)\\ &= 32 * (¼)\end{aligned}$$

Thus, thirty-two quarter beats are required to generate total length of rhythm for 8 bars with 4/4 time signature.

When the rhythm generator receives a time signature from the user, it calculates beat duration ($1/(2^n * V)$) and total number of beats $2^n(B* Bt)$ by taking the random value of n.

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

Algorithm 3.1 Bresenham's Line Algorithm for Rhythm
Input: Start point (x_1, y_1), End point (x_2, y_2)
Output: Binary Sequence (S)

```
1: Enter the value of (x1, y1), (x2, y2);
2: Consider (x, y) as the starting point, x = x1, y = y1;
3: Calculate dx = x2 - x1, dy = y2 - y1;
4: Calculate d = 2 * dy - dx;
5: Check if x >= x2, then return S and stop;
6: If (d < 0) then d = d + 2dy, Append 0 to S;
7: If (d >= 0) then d = d + 2dy - 2dx, y = y + 1, Append 1 to S;
8: Increment x = x + 1;
9: Go to step 5;
```

By doing so, we can generate different beat duration for the same time signature and bring variety in rhythm. The number of beats calculated above is passed in Algorithm 3.1 to initialise the endpoints. The slope of the line is taken from 0 to 1. The endpoints of the line are initialised as follows:

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

The algorithm generates binary sequence, as we can see in Figure 3.3. Each bit denotes the beat duration ($1/(2^n * V)$). Bit value 1 implies the beginning of a new note ,while bit value 0 implies continuation of previous note. Figure 3.4 shows conversion of binary sequence to note duration.

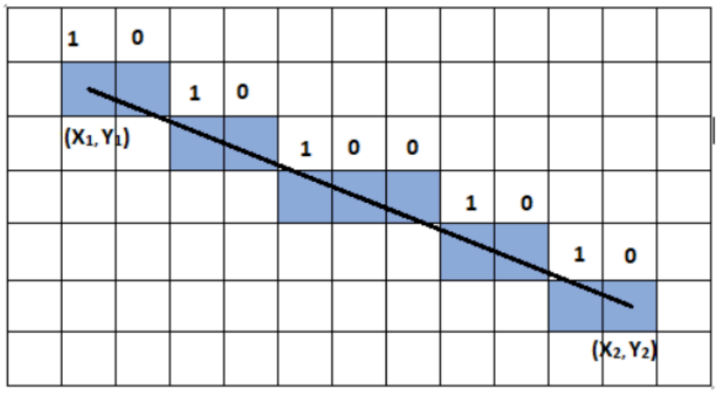


Figure 3.3: Example of output binary sequence with (x_1, y_1) and (x_2, y_2) as input.

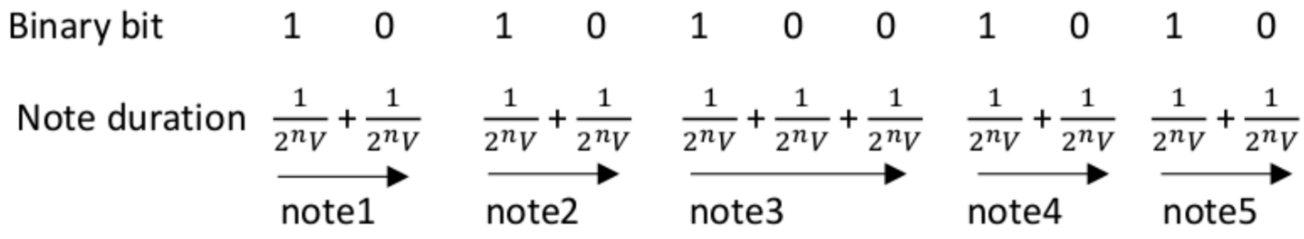

Binary bit
1 0 1 0 1 0 0 1 0 1 0
Note duration
$\frac{1}{2^n V} + \frac{1}{2^n V}$
$\frac{1}{2^n V} + \frac{1}{2^n V}$
$\frac{1}{2^n V} + \frac{1}{2^n V} + \frac{1}{2^n V}$
$\frac{1}{2^n V} + \frac{1}{2^n V}$
$\frac{1}{2^n V} + \frac{1}{2^n V}$
note1
note2
note3
note4
note5

Figure 3.4: Example of converting a binary sequence to note duration.

In this way, we have obtained the number of notes and duration of each note. 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.

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.

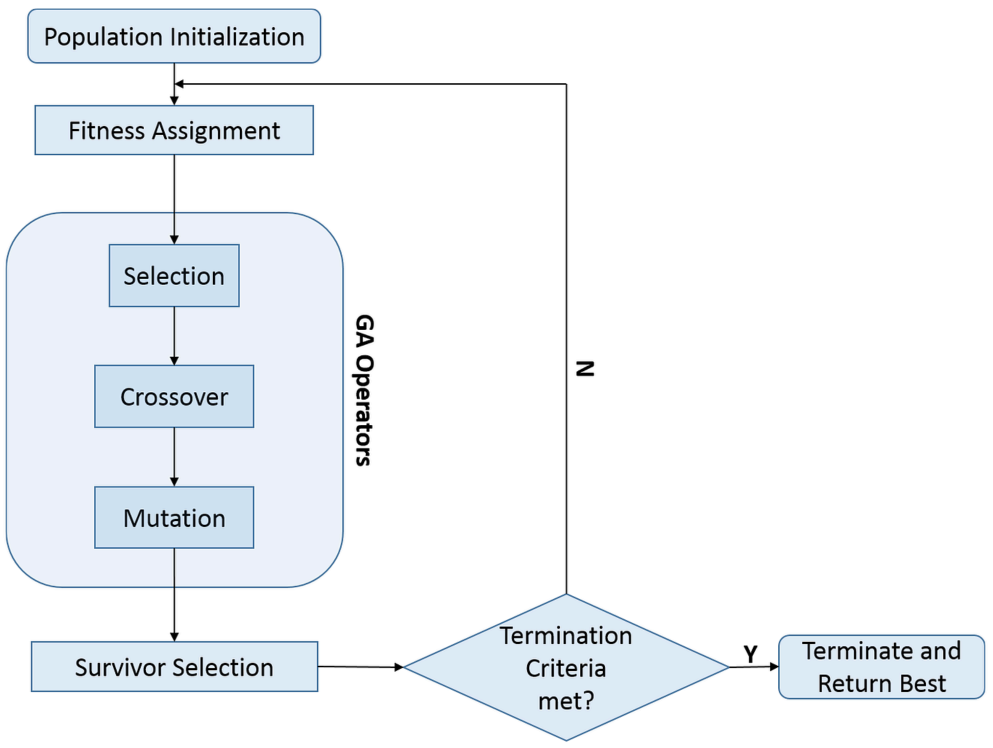
Step 6: Mutation
Introduce small, random changes to the offspring's genetic information. Mutation adds diversity to the population and helps explore new regions of the solution space.

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.



Termination criteria in this case is a piece of music which sounds good (that is music with good melody, rhythm and harmony)

MELODY GENERATION USING GENETIC ALGORITHM

Melody: Melody is a sequence of single pitches that are organised and arranged to create a musical line. It is an important element of music, and it is often what listeners remember most about a piece of music.

Step 1: Encoding Type = <Note index, Octave index, Binary bit>
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 numbers 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 the user's motif while bit '0'(default value) is appended to the randomly generated notes. (Figure 3.5)

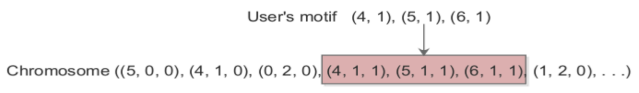

User's motif (4, 1), (5, 1), (6, 1)
Chromosome ((5, 0, 0), (4, 1, 0), (0, 2, 0), (4, 1, 1), (5, 1, 1), (6, 1, 1), (1, 2, 0), . . .)

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

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 from <2, 0, 1> <4, 1, 1> <1, 2, 1> <5, 1, 0> <3, 0, 1> <7, 0, 0> <0, 1, 1> <6, 2, 1>)

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, 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 the chromosome.

$$Ab = OI * S + NI \tag{3.2}$$

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

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

  where Ab_i is the absolute value of ith 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 the 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 penalises the genome having consecutive note repetition. Penalty is equal to the maximum consecutive occurrence of a note within a chromosome.

HARMONY GENERATION USING GENETIC ALGORITHM

Harmony: The study of harmony involves the juxtaposition of individual pitches to create chords, and in turn the juxtaposition of chords to create larger chord progressions.

Step 1: Encoding Type = <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 from <2, 0, 1, 0> <4, 1, 1, 2> <1, 2, 1, 3> <5, 1, 0, 2> <3, 0, 1, 1> <7, 0, 0, 0> <0, 1, 1, 1> <6, 2, 1, 3>)

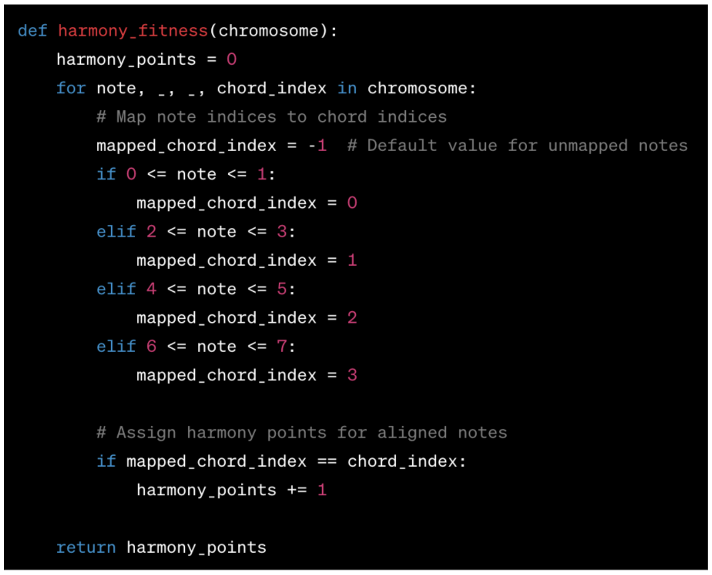
Step 4: Fitness Function
Fitness = w1*ON + w2*TF + w3*NJ + w4*RN + w5*HF + w6*CP
= 1*2 + 1*2 + 1*1 + 1*0 + 1*3 + 1*3 = 11

Harmony Fitness (HF):
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.

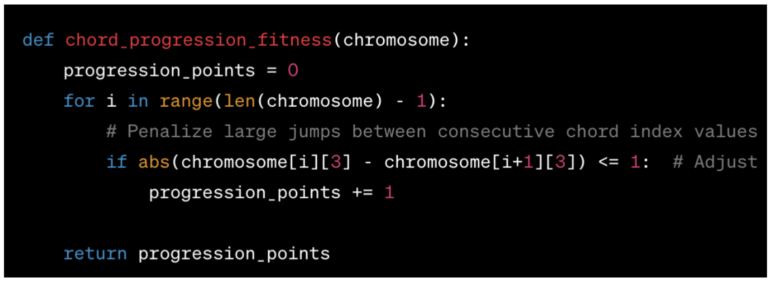
For note indices 4 and 5, the mapped chord index is 2.
For note indices 6 and 7, the mapped chord index is 3.

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.



```
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
```

Chord Progression (CP):
The chord_progression_fitness function evaluates the quality of chord progressions in the melody. It awards points for smooth transitions between consecutive chords and penalises 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.



```
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
```

FIRST ITERATION OF GENETIC ALGORITHM

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

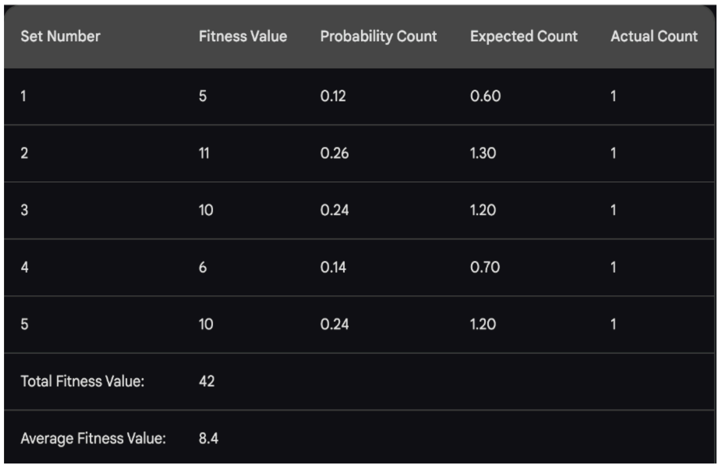
Step 1: Encoding Type = <Note index, Octave index, Binary bit, Chord Index>

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)

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.

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

Not taking into account - User Motif.

1. ***bresenhams_line_algo.py*** *(Code for rhythm generation)*

```
import random
import math
# Function to calculate beat parameters based on time signature
def calculate_beat_parameters(time_signature):
   top_number, bottom_number = map(int, time_signature.split('/'))
   Bt = top_number  # Beats per bar
   V = bottom_number  # Beat value
   B = 8  # Number of bars
   # Calculate beat duration and total number of beats
   min_val = math.log2(2/V)
   max_val = math.log2(16/V)
   n = random.randint(math.ceil(min_val), math.floor(max_val))
   print("n:", n)
   note_duration = 1 / ((2 ** n) * V)
   print("Note duration: ", 1/note_duration)
   total_beats = (2 ** n) * (B * Bt)
   return total_beats, note_duration
# Function to generate binary sequence using Bresenham's line algorithm
def bresenhams_line_algorithm(x1, y1, x2, y2):
   number_of_notes = 0
   x = x1
   y = y1
   dx = x2 - x1
   dy = y2 - y1
   d = 2 * (dy - dx)
   binary_sequence = []
   while x < x2:
       if d < 0:
           d += 2 * dy
           binary_sequence.append(0)
       else:
           d += 2 * (dy - dx)
           y += 1
           binary_sequence.append(1)
           number_of_notes += 1
```

```
        x += 1
    return binary_sequence, number_of_notes
```

Function returns the binary sequence and calculates number of notes based on the sequence.

2. ***genetic_algo_fitness.py*** *(Code for melody and harmony generation)*

```
import random
# Function to generate initial population for genetic algorithm
def generate_initial_population(number_of_notes, num_chromosomes, scale, range_,
user_motif):
    population = []
    for _ in range(num_chromosomes):
        chromosome = []
        for _ in range(number_of_notes):
            # Randomly initialize each tuple in the chromosome
            note_index = random.randint(0, len(scale) - 1)
            octave_index = random.randint(0, len(range_) - 1)
            binary_bit = user_motif
            chord_index = random.randint(0, 3)
            chromosome.append((note_index, octave_index, binary_bit, chord_index))
        population.append(chromosome)
    return population
# Function to calculate Octave Note Fitness (ON)
def octave_note_fitness(chromosome):
    octave_counts = {}
    for note_tuple in chromosome:
        octave_index = note_tuple[1]  # Octave index is the second element in the tuple
        if octave_index in octave_counts:
            octave_counts[octave_index] += 1
        else:
            octave_counts[octave_index] = 1
    return max(octave_counts.values(), default=1)
# Function to calculate Triad Fitness (TF)
def triad_fitness(chromosome, scale):
    tf = 2
    for i in range(len(chromosome) - 2):
        abi = abs(chromosome[i][1] * len(scale) + chromosome[i][0])  # Absolute value
of note index
        abi_plus_1 = abs(chromosome[i + 1][1] * len(scale) + chromosome[i + 1][0])
        abi_plus_2 = abs(chromosome[i + 2][1] * len(scale) + chromosome[i + 2][0])
```

```
       if (abi < abi_plus_1 < abi_plus_2) or (abi > abi_plus_1 > abi_plus_2):
           tf += 1
   return tf
# Function to calculate No Jump Fitness (NJ)
def no_jump_fitness(chromosome, scale):
   nj = 1
   for i in range(len(chromosome) - 1):
       abi = abs(chromosome[i][1] * len(scale) + chromosome[i][0])  # Absolute value
of note index
       abi_plus_1 = abs(chromosome[i + 1][1] * len(scale) + chromosome[i + 1][0])
       abii = abs(abi - abi_plus_1)
       if abii < 4:  # Considering perfect fourth interval
           nj += 1
   return nj
# Function to calculate Repeat Note Fitness (RN)
def repeat_note_fitness(chromosome):
   max_consecutive_repeats = 0
   consecutive_repeats = 0
   for i in range(len(chromosome) - 1):
       if chromosome[i][0] == chromosome[i+1][0]:
           consecutive_repeats += 1
           max_consecutive_repeats = max(max_consecutive_repeats, consecutive_repeats)
       else:
           consecutive_repeats = 1
   return max_consecutive_repeats
# Function to calculate Harmony Fitness (HF)
def harmony_fitness(chromosome):
   chord_indices = [0, 0, 1, 1, 2, 2, 3, 3]  # Chord indices mapped to note indices
   hf = 0
   for note_tuple in chromosome:
       note_index = note_tuple[0]
       mapped_chord_index = chord_indices[note_index]
       if note_tuple[3] == mapped_chord_index:
           hf += 1
   return hf
# Function to calculate Chord Progression (CP)
def chord_progression_fitness(chromosome):
   cp = 0
   for i in range(len(chromosome) - 1):
       if abs(chromosome[i][3] - chromosome[i+1][3]) <= 1:
           cp += 1
   return cp
```

```
# Function to calculate overall fitness using given weights
def calculate_overall_fitness(chromosome, scale, weights):
   on_fitness = octave_note_fitness(chromosome)
   tf_fitness = triad_fitness(chromosome, scale)
   nj_fitness = no_jump_fitness(chromosome, scale)
   rn_fitness = repeat_note_fitness(chromosome)
   hf_fitness = harmony_fitness(chromosome)
   cp_fitness = chord_progression_fitness(chromosome)
   overall_fitness = ((weights[0] * on_fitness) + (weights[1] * tf_fitness) +
(weights[2] * nj_fitness)
                    + (weights[3] * rn_fitness) + (weights[4] * hf_fitness) +
(weights[5] * cp_fitness))
   return overall_fitness
```

Function returns the overall fitness values for all genomes in the randomly generated population.

3. ***selection.py*** *(Code for Roulette Wheel Selection for parents)*

```
   def roulette_wheel_selection(population, fitness_values):
   total_fitness = sum(fitness_values)
   normalized_probabilities = [fitness / total_fitness for fitness in fitness_values]
   # Sort the normalized probabilities and zip them with the population
   sorted_probabilities_and_population = sorted(zip(normalized_probabilities,
population, fitness_values), reverse=True)
   # Select the top 2 parents
   selected_parents = [chromosome for probability, chromosome, fitness in
sorted_probabilities_and_population[:2]]
   selected_fitness_values = [fitness for probability, chromosome, fitness in
sorted_probabilities_and_population[:2]]
   return selected_parents, selected_fitness_values
```

Function returns top 2 selected parents and their fitness values.

4. ***crossover_and_mutation.py*** *(Code for crossover and mutation of selected parents)*

```
return random
def calculate_melodic_interval(parent1, parent2):
   melodic_intervals = []
   for i in range(len(parent1)):
       Ab1_k = abs(parent1[i][1] * 8 + parent1[i][0])
       Ab2_k_plus_1 = abs(parent2[(i + 1) % len(parent2)][1] * 8 + parent2[(i + 1) %
len(parent2)][0])
       Ab2_k = abs(parent2[i][1] * 8 + parent2[i][0])
       Ab1_k_plus_1 = abs(parent1[(i + 1) % len(parent1)][1] * 8 + parent1[(i + 1) %
len(parent1)][0])
       melodic_interval = max(abs(Ab1_k - Ab2_k_plus_1), abs(Ab2_k - Ab1_k_plus_1))
       melodic_intervals.append(melodic_interval)
   return melodic_intervals
def crossover(parent1, parent2):
   melodic_intervals = calculate_melodic_interval(parent1, parent2)
   min_interval_index = melodic_intervals.index(min(melodic_intervals))
   child1 = parent1[:min_interval_index] + parent2[min_interval_index:]
   child2 = parent2[:min_interval_index] + parent1[min_interval_index:]
   return child1, child2, melodic_intervals[min_interval_index], melodic_intervals
#function for mutation
def note_based_mutation(chromosome):
   mutated_chromosome = chromosome[:]  # Make a copy of the chromosome
   for i in range(len(mutated_chromosome)):
       # Get the current note tuple
       current_note = mutated_chromosome[i]
       current_note_number = current_note[0]  # Calculate the note number
       # Check the neighboring notes if their difference is greater than 4
       if i > 0:
           # Calculate the note number of the previous note
           previous_note = mutated_chromosome[i - 1]
           previous_note_number = previous_note[0]
           if abs(current_note_number - previous_note_number) > 4:
               # If the difference is greater than 4, replace the note with a new note
               new_note_index = random.randint(max(0, previous_note_number - 4),
min(7, previous_note_number + 4))
               mutated_chromosome[i] = (new_note_index, current_note[1],
current_note[2], current_note[3])
       if i < len(mutated_chromosome) - 1:
           # Calculate the note number of the next note
           next_note = mutated_chromosome[i + 1]
           next_note_number = next_note[0]
           if abs(current_note_number - next_note_number) > 4:
               # If the difference is greater than 4, replace the note with a new note
               new_note_index = random.randint(max(0, next_note_number - 4), min(7,
next_note_number + 4))
```

```
                mutated_chromosome[i] = (new_note_index, current_note[1],
current_note[2], current_note[3])
   return mutated_chromosome
```

Function returns the mutated chromosome.

5. ***mapping.py** (Code that maps chromosome into <notes, duration> format)*

```
def map_chromosome_to_notes_duration(chromosome, binary_sequence,
note_duration_single):
   notes = []  # List to store note durations
   first_pointer = None  # Pointer for the first occurrence of 1
   second_pointer = None  # Pointer for the second occurrence of 1

   # Find the first occurrence of 1 in the binary sequence
   first_pointer = binary_sequence.index(1)

   # Find the next occurrence of 1 after the first_pointer
   try:
       second_pointer = binary_sequence.index(1, first_pointer + 1)
   except ValueError:
       # If no more occurrences of 1 are found, return an empty list
       return []
   # Loop until there are no more occurrences of 1 after second_pointer
   while second_pointer is not None:
       # Calculate the duration between consecutive 1s
       duration = (second_pointer - first_pointer) * note_duration_single
       # Append the duration to the notes list
       notes.append(duration)
       # Move the first_pointer to the position of the second_pointer
       first_pointer = second_pointer
       # Find the next occurrence of 1 after the second_pointer
       try:
           second_pointer = binary_sequence.index(1, first_pointer + 1)
       except ValueError:
           # If no more occurrences of 1 are found, set second_pointer to None
           second_pointer = None
   # If the loop ends and there's one more note after the last occurrence of 1
   if second_pointer == None:
       duration = (len(binary_sequence) - first_pointer) * note_duration_single
       notes.append(duration)
   return notes
```

```
def convert_chromosome_to_note_number(chromosome, note_duration):
    # Define the mapping between chromosome values and note numbers
    note_mapping = {
        (0, 0, 0, 0): 60, (0, 0, 0, 1): 61, (0, 0, 0, 2): 67, (0, 0, 0, 3): 63,
        (0, 1, 0, 0): 72, (0, 1, 0, 1): 73, (0, 1, 0, 2): 79, (0, 1, 0, 3): 75,
        (0, 2, 0, 0): 84, (0, 2, 0, 1): 85, (0, 2, 0, 2): 91, (0, 2, 0, 3): 87,
        (1, 0, 0, 0): 62, (1, 0, 0, 1): 63, (1, 0, 0, 2): 65, (1, 0, 0, 3): 65,
        (1, 1, 0, 0): 74, (1, 1, 0, 1): 75, (1, 1, 0, 2): 77, (1, 1, 0, 3): 77,
        (1, 2, 0, 0): 86, (1, 2, 0, 1): 87, (1, 2, 0, 2): 89, (1, 2, 0, 3): 89,
        (2, 0, 0, 0): 64, (2, 0, 0, 1): 68, (2, 0, 0, 2): 67, (2, 0, 0, 3): 67,
        (2, 1, 0, 0): 76, (2, 1, 0, 1): 80, (2, 1, 0, 2): 79, (2, 1, 0, 3): 79,
        (2, 2, 0, 0): 88, (2, 2, 0, 1): 92, (2, 2, 0, 2): 91, (2, 2, 0, 3): 91,
        (3, 0, 0, 0): 65, (3, 0, 0, 1): 66, (3, 0, 0, 2): 64, (3, 0, 0, 3): 64,
        (3, 1, 0, 0): 77, (3, 1, 0, 1): 78, (3, 1, 0, 2): 76, (3, 1, 0, 3): 76,
        (3, 2, 0, 0): 89, (3, 2, 0, 1): 90, (3, 2, 0, 2): 88, (3, 2, 0, 3): 88,
        (4, 0, 0, 0): 67, (4, 0, 0, 1): 68, (4, 0, 0, 2): 66, (4, 0, 0, 3): 66,
        (4, 1, 0, 0): 79, (4, 1, 0, 1): 80, (4, 1, 0, 2): 78, (4, 1, 0, 3): 78,
        (4, 2, 0, 0): 91, (4, 2, 0, 1): 92, (4, 2, 0, 2): 90, (4, 2, 0, 3): 90,
        (5, 0, 0, 0): 69, (5, 0, 0, 1): 70, (5, 0, 0, 2): 64, (5, 0, 0, 3): 64,
        (5, 1, 0, 0): 81, (5, 1, 0, 1): 82, (5, 1, 0, 2): 76, (5, 1, 0, 3): 76,
        (5, 2, 0, 0): 93, (5, 2, 0, 1): 94, (5, 2, 0, 2): 88, (5, 2, 0, 3): 88,
        (6, 0, 0, 0): 71, (6, 0, 0, 1): 67, (6, 0, 0, 2): 66, (6, 0, 0, 3): 66,
        (6, 1, 0, 0): 83, (6, 1, 0, 1): 79, (6, 1, 0, 2): 78, (6, 1, 0, 3): 78,
        (6, 2, 0, 0): 95, (6, 2, 0, 1): 91, (6, 2, 0, 2): 90, (6, 2, 0, 3): 90,
    }
    # Initialize an empty list to store the notes
    notes = []
    # Iterate through the chromosome and convert to note number and duration
    for gene in chromosome:
        note_number = note_mapping[gene]
        notes.append(note_number)
    return notes
```

6. ***midi_generation.py*** *(Code for generating output midi file)*

```
import mido
from mido import MidiFile, MidiTrack, Message, MetaMessage
def notes_to_midi(notes, output_file='output.mid', tempo=50000):
    mid = MidiFile()
    track = MidiTrack()
    mid.tracks.append(track)
    # Add tempo meta message
    track.append(MetaMessage('set_tempo', tempo=tempo))
    ticks_per_beat = mid.ticks_per_beat
    for note in notes:
        note_number, duration = note
        ticks = int((duration * ticks_per_beat) / (tempo / 1000000))
        track.append(Message('note_on', note=note_number, velocity=64, time=0))
        track.append(Message('note_off', note=note_number, velocity=64, time=ticks))
    mid.save(output_file)
    print(f"MIDI file saved as {output_file}")
```

7. ***main.py** (Main code that integrates all the functionality to generate desired output)*

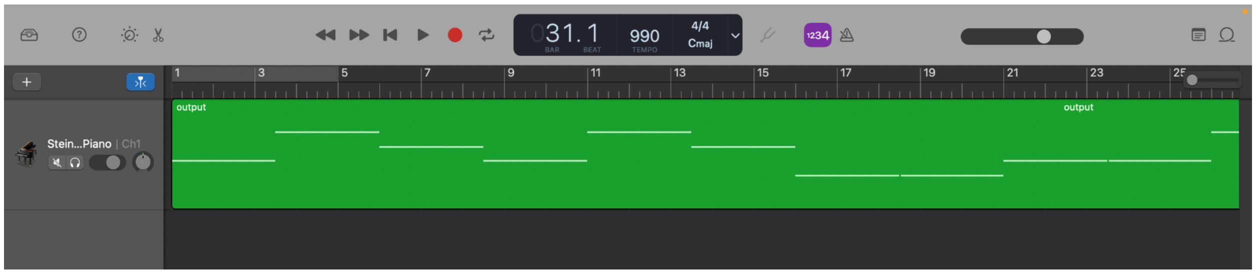
```
import random
from bresenhams_line_algo import calculate_beat_parameters, bresenhams_line_algorithm
from genetic_algo_fitness import generate_initial_population, octave_note_fitness,
triad_fitness, no_jump_fitness, repeat_note_fitness, harmony_fitness,
chord_progression_fitness, calculate_overall_fitness
from selection import roulette_wheel_selection
from crossover_and_mutation import calculate_melodic_interval, crossover,
note_based_mutation
from mapping import map_chromosome_to_notes_duration,
convert_chromosome_to_note_number
from midi_generation import notes_to_midi

def main():
    # Parameters
    time_signature = '1/4'
    scale = ['C', 'D', 'E', 'F', 'G', 'A', 'B']
    range_ = list(range(3, 5))  # Octaves
    num_chromosomes = 50
    user_motif = 0
    weights = [1, 1, 1, -1, 1, 1]
    # Step 1: Calculate beat parameters
    total_beats, note_duration = calculate_beat_parameters(time_signature)
    # Step 2: Initialize endpoints for Bresenham's algorithm
    x1, y1 = 0, 0
    x2 = total_beats
    y2 = random.randint(0, total_beats)  # Randomize y2 within the range of total beats
    # Step 3: Perform Bresenham's line algorithm
    binary_sequence, number_of_notes = bresenhams_line_algorithm(x1, y1, x2, y2)
    # Step 4: Initialize population for genetic algorithm
    population = generate_initial_population(number_of_notes, num_chromosomes, scale,
range_, user_motif)
    # Step 5: Calculate fitness of each chromosome in the population
    fitness_values = []
    for chromosome in population:
        fitness = calculate_overall_fitness(chromosome, scale, weights)
        fitness_values.append(fitness)
      # Step 6: Perform roulette wheel selection to select top 2 parents
    selected_parents, selected_parents_fitness = roulette_wheel_selection(population,
fitness_values)
    # Step 7: Apply crossover and mutation to both the childs obtained
    child1, child2, melodic_interval, melodic_intervals_array =
crossover(selected_parents[0], selected_parents[1])
    mutated_child1 = note_based_mutation(child1)
    mutated_child2 = note_based_mutation(child2)
    mutated_child1_iterated = mutated_child1
    mutated_child2_iterated = mutated_child2
    #iteration
    for _ in range(6):
        child1_iterated, child2_iterated, melodic_interval, melodic_intervals_array =
crossover(mutated_child1_iterated, mutated_child2_iterated)
        mutated_child1_iterated = note_based_mutation(child1_iterated)
        mutated_child2_iterated = note_based_mutation(child2_iterated)
    mutated_child1 = mutated_child1_iterated
    mutated_child2 = mutated_child2_iterated
    # Step 8: Mapping to convert chromosomes into (note_number, note_duration) format
    note_duration_mapped1 = map_chromosome_to_notes_duration(mutated_child1,
binary_sequence, note_duration)
    note_number_mapped1 = convert_chromosome_to_note_number(mutated_child1,
note_duration)
    notes_array1 = list(zip(note_number_mapped1, note_duration_mapped1))
    note_duration_mapped2 = map_chromosome_to_notes_duration(mutated_child2,
binary_sequence, note_duration)
    note_number_mapped2 = convert_chromosome_to_note_number(mutated_child2,
note_duration)
    notes_array2 = list(zip(note_number_mapped2, note_duration_mapped2))
    final_array = notes_array1 + notes_array1
    final_array += notes_array2
    final_array += notes_array1
    # Step 9: Midi generation
    notes_to_midi(final_array, 'Final/output.mid')
    print("Binary Sequence: ", binary_sequence)
    print("Number of notes: ", number_of_notes)
    print("Note duration: ", note_duration)
    print("Initial Population:", population)
    print("Fitness values:", fitness_values)
    print("Selected Parents:", selected_parents)
    print("Fitness values of selected parents:", selected_parents_fitness)
    print("Crossover Point:", melodic_interval)
    print("Melodic Interval Array:", melodic_intervals_array)
    print("Parent 1 before crossover:", selected_parents[0])
    print("Parent 2 before crossover:", selected_parents[1])
    print("Child 1 after crossover:", child1)
    print("Child 2 after crossover:", child2)
    print("Child 1 after mutation:", mutated_child1)
    print("Child 2 after mutation:", mutated_child2)
    print("Final Notes Array:", final_array)
if __name__ == "__main__":
    main()
```

RESULTS AND DISCUSSIONS

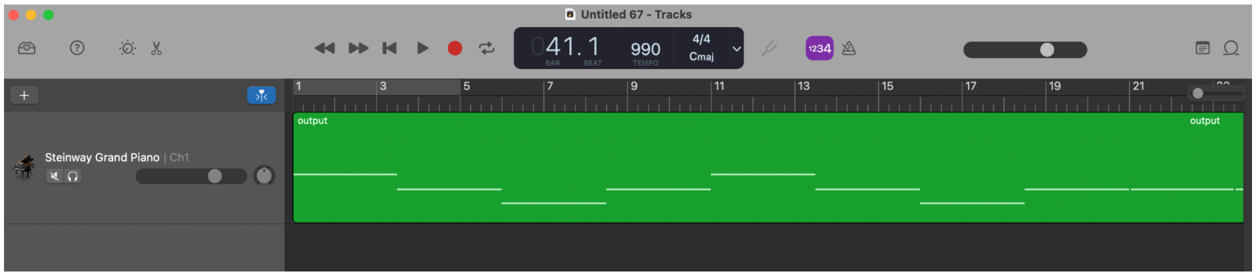
- **INPUT**

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

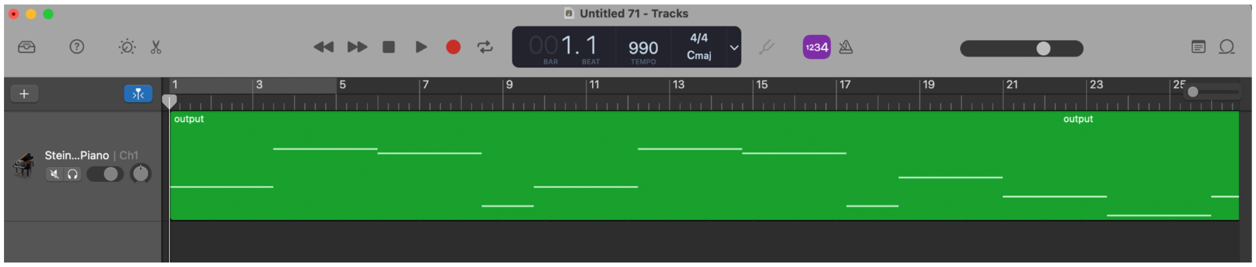
- **OUTPUT**



output_1.mid



output_2.mid


Untitled 71 - Tracks
001. 1
BAR
BEAT
990
TEMPO
4/4
Cmaj
1234
Stein...Piano | Ch1
output
output
1
3
5
7
9
11
13
15
17
19
21
23
25

output_3.mid

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 optimise algorithm parameters to encourage more diverse and creative musical outputs.
    - Benefits: Allows for a nuanced control over the algorithm's behaviour, 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

The integration of genetic algorithms with harmony components has led to compositions characterised by heightened melodic richness and enhanced harmonic engagement. This innovative fusion represents a significant advancement in music composition, where computational techniques intertwine with artistic expression. Through successive iterations, the algorithm has demonstrated remarkable evolutionary capacity, consistently refining its musical output and showcasing inherent adaptability in its design.

Significant progress has been achieved through deliberate adjustments to algorithm parameters, resulting in tangible improvements in the overall musical quality. This positive trend indicates successful enhancement in the generated music's quality, highlighting the algorithm's potential for fostering evolutionary progress. The exploration of genetic algorithms in music composition not only highlights their computational efficiency but also unveils a promising avenue for creative expression.

The interplay between algorithmic precision and artistic ingenuity is evident in the evolving landscape of musical creation. The algorithm's adeptness in navigating and adapting to diverse musical styles suggests a symbiotic relationship between technology and artistry. This harmonious fusion unlocks new possibilities for composers and musicians, providing a unique tool that enhances their creative process. As we delve deeper into the synthesis of computational methods and artistic vision, the horizon of musical expression expands, paving the way for novel compositions that seamlessly blend the best of both worlds.