A Model of Equal-Temperament Tunings

Reed Axewielder
Entrepreneur Extraordinaire
AXEWIELDER GOODS
(Guillermo M. Dávila Andino)

May 24, 2025

Abstract

With this enterprise, we strive to arrive at a Distance metric between heptatonic scales with which we can measure the complexity of modulations and, maybe find an application of metric spaces to harmonic modulation. Bottom line, our main motivation for undertaking this endeavor is to have a tool, a framework, a system to aid in the process of musical composition. Now, we have a complete model of Equal Temperament Tuning, and therefore, a Complete theory of harmony in such tunings.

Contents

1 Background

In an attempt to fabricate the music we wanted to hear, we found ourselves in the need for an objective means of describing the effect of any musical phenomena. Thus tooled, we may weigh the musical value of each one of them and confidently inform our compositional decisions. In music, a composer is given free reign, but if they are in views of an effect, they'd better know why. Many people go to all kinds of different schools, just to learn some music. Well, here, I propose a Model of Equal Temperament Tunings. Regardless of the number of tones in a tuning of Equal Temperament, and regardless of the reference note, this Model aims to systematically represent every harmonic entity possible in all Equal Temperament Tunings.

Nevertheless for most illustrative purposes, our tuning of preference will be Twelve Tone Equal Temperament Tuning for $A_4 = 440$Hz. (12-TET)

1.1 History

Even before Equal Temperament (ET) was even dreamed about, humans have already been playing music. But the motivations behind the development of ET say more about our interests as humans than about what music is. We would rather have a system open to collaboration that is easy to reproduce than worry about the ratios within your harmonies or whatnot. So, Equal Temperament is a response to a human desire to communicate. A solution that brings a different set of problems or challenges. Like, agreeing in which pitch to assign our reference note, or having to come to terms with the fact that the Harmonic Sequence is not the only basis by which to order, categorize, classify or measure the different musical ideas and phenomena available.

The history of the development of ET has been well treated by others before. So we will skip over to more relevant matters.

Allen Forte and others laid the foundation for what is called Set Theory in music. While instructive toward the same direction as our enterprise, we found the need for more time to study their material at length.[7] At least, we agree on the number of unique scale definitions being 351. Equivalent to the total of Forte's Numbers. We disagree on the numerical value of our pitches in *The Strucure of Atonal Music*,

> The integer 0 has been assigned to C [...], and so on until the integer 11 has been assigned to B, completing the octave. [...] This assignment remains fixed throughout the present volume.[7, p. 3]

Morever, his notation is not intuitive for me either as a musician or as a mathematician. He really never goes deep into the issue of ordering pitch classes.

In *Basic Atonal Theory*, John Rahn Clearly states that

> It does *not* follow that, because we are using integers to name pitches [...], all those things that are true of integers are going to be true of pitches [...]. We must carefully determine the limits of similarity between integers [...] and pitches [...]. To do otherwise would be to fall into the *numerological fallacy*. [8, p. 19]

It seems to me that he just doesn't want to be burned at the stake for witchcraft. But the truth of the matter is that he, from the outset, refuses to properly align the set of pitches with the integers, and in doing so, misses an oportunity to make his own life easier. If we believe that all was created by Our Lord, God Almighty, and that everything comes from Him, we must accept that we must be able to find the origins and the nature of the things we observe. If we believe that God created everything in good order, we must accept that He must have had created musical ideas in good order as well.

At least, we agree on the ordering of intervals. Now, he defines the "normal form" of a set of pitches that is similar to ours in a very subtle manner.

> The "normal form" of a set is that ordering of its members which is increasing within an octave and most packed to the left; if there is more than one such ordering, it is the remaining ordering with the smallest initial pc number.

What for Rahn is merely a notion of being most "packed to the left", we replace with our definition of the concentration or diffusion of a scale, and the order we give is that of the natural production of the permutations of the partitions of the densities of the scales in ET.

He spends a great deal in transpositions and inversions. We give a simple implementation of the transpose function. (8.2, lns. 340 to 343.)

Joseph N. Strauss assigns C to 0 arbitrarily and thinks no more about it.

> Example 1-6 shows the twelve pitch classes [...], following a "fixed *do*" notation: the pitch class containing the Cs is arbitrarily assigned the integer 0, and the rest follow from there.[9, section 1.4, p. 5]

He also gives a list of names for different intervals,[9, Example 1.10, p. 8] defines his *normal form* as "the most compressed way of writing a pitch-class set", explains a set of rules on how he goes about this and clarifies that his method is somewhat different from "the more traditional normal form".[9, section 2.2, p. 45]

From what I see, these authors fail to acknowledge the intrinsic order that the interval induces into the category of fundamental scale definitions. It is for the very leading tone that the harmonic and melodic minor scales were devised!

Xenakis had previously made use of computer programming to generate sounds, using FORTRAN IV.[4] But he focused heavily on stochasticism. Even though our approach is combinatorial, meaning it's an interpretation that lends itself well to probabilistic and statistical models, it is not clear at the moment whether our applications of these theories are even compatible. There's the THX Deep Note glissando, which was also generated digitally by means of computer programming.

Miller Puckette, among all the people who've undertaken the task of creating a computer language for music, has created at least two! He's also made publicly available a book with the necessary ideas behind digital synthesis of sound.[2]

Rich Cochrane has a set of books. One of which I found greatly inspiring, since many other books about scales would just repeat the same few 14 or less almost everybody uses in the different keys. It offers a complete account of the scales possible in 12-TET tuning, but not as much as a definitive order within families of , or even a method to logically address them.[3]

1.2 Practice

Different tuning systems have been devised all over the world. But the Western world settled on 12 Tone Equal Temperament with A_4 at 440Hz considered the most usual reference tone by default in many softwares. With the development of accessible computational machines, it is our responsibility to find spaces of intersection between technology and the arts. Much vocabulary has been developed popularly and scholarly to refer to musically related concepts. Words like form, orchestration, melody, note, progression, chord, tempo, rhythm, etc. We will provide our own definitions when necessary. But many of these deserve their own treatment, as Harmonic Progression is just one of many aspects in the conception of an overarching system for composing music that we hope to complete sometime, in the future, for God's Glory! (Many of these concepts may reach beyond the scope of any particular tuning system anyway.)

2 Notions

Let's start with the notion of a scale. We have twelve equally distributed tones within one octave in our tuning. Each tone determines a certain pitch class.

When we select, out of all the pitch classes, a subset of these, we call that a scale. Each note of each scale heads a mode of the scale. When, out of any scale, you select a subset of pitch classes, we call these chords. Scales are either symmetric or antisymmetric. If a mode of a scale is enharmonic to any other within that scale, then it is said to be symmetric. Otherwise an antisymmetric scale will always have exactly as many modes as the scale's constituent pitch classes.

To begin our journey, we need a little bit of Prime Matter. This will be our universe of heptatonic scales. The most famous and common of which is the Diatonic scale. And for good reason! There is much discourse we can produce about this scale. For starters, it's the only non-reversible, asymmetric heptatonic scale in 12-TET that has no more than two consecutive notes and whose notes are separated by no more than two half-steps. By non-reversible, we mean that the scale generated by applying the intervals of the definition of the Diatonic scale in reverse order spells out a different scale or mode. We say it's asymmetric, because none of its modes are enharmonic to each other. Which means they're all spelled differently. When we relax our definition of the Diatonic scale in different ways we find other scales. By allowing the notes to be separated by, at most, a whole-step and a half, we find the Harmonic Minor. In contrast, by losing the requirement of asymmetry instead, we find the Ascending Melodic Minor. And so on. But this is not why we're here. The subject has also been well treated by others before.

Whereas, previously, Schoenberg and others derive the diatonic scale from the harmonic series, we treat the modern tuning as a starting point in itself. The notes being equally spaced within the octave is a crucial fact for our investigation, since the diatonic scale is the only heptatonic scale whose notes are maximally spread, among other properties.

2.1 Conventions

All scales exhibit a degree of concentration or diffusion. We define a Diatonic Scale as the most diffuse scale of those of as many semitones. So, in our Twelve Tone Equal Temperament Tuning, we have exactly twelve Diatonic Scales. The first note of any scale will always be zero. This way are all our scales normalized. When dealing with the Heptatonic Diatonic Scale in C, we assign the letter B to the zeroth tone. The next note of the scale will be that of the first interval of the scale. In this case, a minor second, from B to C, of a total of one semitone. So, we assign the letter C to the first tone. Therefore, A is the tenth tone of the chromatic scale.

2.2 Philosophy

It follows from the fact that the natural numbers are constructed from the empty set towards an invariant direction that we liken to an axis, obviously, from a particular position. Negative integers are natural numbers constructed in a direction most opposite that of the naturals, from the same position also. Therefore, since that direction can be ascertained, as so can many others, we can do a particular kind of algebra that allows us to perform certain kinds of operations over these scales and their aggregates and derivatives. You could have, from the same starting point (a pitch reference), different directions, or different intervals that spell out different scales. If we order all the intervals from any reference pitch, we can have use the same order of intervals for any other pitch. Therefore, if we order all the scales from an arbitrary reference pitch, all the same scale definitions could apply for any other Key or Tonic.

Silence is another kind of interpretation of zero, in terms of emptiness. Emptiness of vibration, or of sound or of meaning, if you get too poetic about it.

We shall use a combinatorial approach to arrive at our solutions.

Since our smallest primitive interval in 12-TET is the semitone, we select it as the unit of measure in our space.

3 Problems

3.1 The Integer Partition Algorithm

Consider any positive integer. The question is, in how many ways can we part that integer. There are various ways to order integer partitions, as stated by Zoghbi. But, since we want to order our scales by number of tones, we look into the particular problem of parting an integer N into n parts. Now, in our code, regardless of the way I would have produced the partitions, the std::set type that contains our std::vector's of partitions in C++ automatically orders its members. When we think about scales in 12-TET, what we really want is to find in how many ways we can divide the octave into different intervals, and having divided our octave into 12 equally distant semitones, we need to know in how many ways we can part 12 into $n \leq 12$ parts.

3.2 Definition of Equal Temperament

Miller Puckette gives the Pitch/Frequency Conversion formulas to "convert between a MIDI pitch m and a frequency in cycles per second f:" [2, 1.4, p. 7]

- $m = 69 + 12 * \log_2(f/440)$
- $f = 440 * 2^{(m-69)/12}$

He uses MIDI's numbering of pitches. For A_4, $m = 69$. Our numbering is somewhat different, as implemented: (8.2, lns. 652 to 663.)

- $n(p, o) = 40 + 12 * (o - 4) + (p - 1)$
- $f(n) = 440 * 2^{\frac{n-49}{12}}$

For A_4, we get $n = 49$ after evaluating the NoteNumber function, for $p \in \mathbb{Z}^+_{12}$ a pitch-class, and $o \in \mathbb{Z}$ any desired octave. We have $R_{C_4} = 40$ be the *Reference NoteNumber* of C_4, $P_{C_4} = 1$, the *Pitch Class* number of C_4, and $O_{C_4} = 4$, the octave of C_4. The pitch-class, octave and value of the tuning's Reference NoteNumber $n(1, 4) := R_{C_4} = 40$ can be can be adjusted for personal preference, particular implementation, etc. This way, in our implementation, we could parametrize the $n(p, o)$ function. More generally, for $R_p = 440$, $R_t = 10$ and $R_o = 4$, being, respectively, the agreed upon *Reference Pitch*, *Reference Tone* and *Reference Octave* of the note A_4, and N being the total number of Tones or pitch-classes in the tuning,

- $n(p, o) = R_{C_4} + N * (o - O_{C_4}) + (p - P_{C_4})$
- $f(n) = R_p * 2^{(n - n(R_t, R_o))/N}$

3.3 What is the space of all scales?

How to determine the order of scales? First, we have the dimension of the number of tones in the scale, from which we determine the *density* of the scale. In 12-TET, a scale may have at most 12 tones. So, all heptatonic scales in 12-TET have a density of 7/12. That's one trivial order. Then we define the most concentrated and least diffuse of scales for each number of tones as that in which all tones are a semitone apart from each other, starting from the zeroth. Then the least diffuse Heptatonic scale is: 0,1,2,3,4,5,6 or in binary notation: 111111100000.

Then, to arrive at the diatonic scale, we find all the integer partitions of 12 into $n \leq 12$ parts and look for all their unique permutations, for some might have multiple internal symmetries. Our partition function can be found in *IntPart.h*. (8.2, lns. 684 to 735.)

Each unique permutation of each integer partition gives a particular scale. Here, we rely on the *std::next_permutation* function from the C++ Standard Libary. Given that this function returns the next permutation in lexicographic order, the first one must be the origin. Having represented the intervals of our pitch classes as integers, we convert it to a *std::string* of characters whose value equals the accorrding integer that represents our interval and use this function to get all the permutations of each given partition, since our partition function spews out the partitions in order of intervals of integer type. Therefore, the partition string that results in the Diatonic Heptatonic scale is: "1 1 2 2 2 2 2".

In general, for N-TET, the set of all scales is generated by finding the unique permutations of all the partitions of N into K parts, for $0 \leq K \leq N$. This gives us a nice property with which, if we know the concentration of a scale of X tones, we can find a scale in Y tones with the same concentration, for $X \neq Y; X, Y \leq N$. Then, if we treat each scale of K notes in N-TET as a vector of K elements in $\mathbb{Z}_N^{+K}$, we can also assign a weight function as the Euclidean Distance between a scale and the zero vector. (8.2, lns. 608 to 619.)

Similarly, we can measure the Euclidean distance between any two vectors in the space of scales to determine which are closest to a vector. This comes in handy when implementing a search algorithm. The next step is to give each scale a Key. We do this simply by element-by-element modular addition of the key value to the normalized scale definition. (8.2, lns. 282 to 304.)

Therefore, assuming our scale is in standard position, (that its first note corresponds to the scale's first mode in whatever key), the Heptatonic Diatonic Scale in 12-TET has the following fundamental definition in binary notation:

[66] 110101101010 or in intervals: 1,2,2,1,2,2,2. This means its diffusion is 66/66, since there are only 66 Heptatonic Scales. In intervals: 0,1,3,5,6,8,10 or "H-W-W-H-W-W-W". Its modes are, in order, Locrian, Ionian, Dorian, Phrygian, Lydian, Mixolydian and Aeolian. In B, for the second mode, we have: {7} ([66]2) 0: 1,3,5,6,8,10,0 = C,D,E,F,G,A,B. (8.2, lns. 313 to 322.)

The last step is to get the Pitch Collection of the tuning. To that effect we use the formula for the equal division of the octave into K parts from the definition of the Equal Temperament Tuning. (8.2, lns. 671 to 677.)

Some of these values might seem arbitrary. They are only parameters or constraints to be fine tuned depending on application, or number of tones. (3.2)

3.4 All the one note modulations between scales?

To begin answering that question, we start with an assumption that the choice of key is irrelevant to the capacity of a scale to move towards a certain direction, and that, because of our choice of temperament, all our scales will behave the same way, and move in the same kinds of directions regardless of the key in which they are evaluated. This means, of course, that there are other aspects, psychological, physiognomical, social, cultural, mechanical, to name a few, that fall beyond this model we're presenting. With that out of the way, we first look for what could be the simplest one note modulations. If we are on the first scale of the one note scales, we have three options:

1. To delete the note into silence.

   ```
   {1} ([1]1) 0    :100000000000: { 0 }
   ([1]1) -> [+0/-0] 1(X) {0} ([0]0)

   {0} ([0]0) 0    :000000000000: { }
   ```

2. To translate the note.

   ```
   ([1]1) -> [+9/-3] 1(9) {1} ([1]1)

   {1} ([1]1) 9    :100000000000: { 9 }
   ```

3. To add a note.

   ```
   ([1]1) -> [+5/-7] A(7) {2} ([5]2)

   {2} ([5]2) 2   :100001000000: { 2, 9 }
   ```

But if we pay close attention, we notice that there's some simple logic underneath. To delete a note may be considered a primitive operation. (8.2, lns. 148 to 173.)

To translate a note involves deleting it and inserting a different note. Regardless, in our implementation to find all the one-note modulations between all scales, we use a method different from the one proposed above. To find which pair of notes need to be interchanged, or swapped, from a given keyed scale, we just call the SetDifference function iterating through the rest of the keyed scales. (8.2, lns. 330 to 339.) If, after the operation, both difference sets contain only one element, we can deduce the translation to be performed. But we need not even have to call our primitive functions to remove notes from and insert notes into scales, as we already know the resulting scale. Nevertheless, to insert a note could be considered another primitive operation. (8.2, lns. 120 to 144.) Then, to add a note is just to insert a note different from the root. When we scale to scales with more than one note, we can:

1. Delete a note from the scale. (8.2, lns. 916 to 962.)

```
{7} ([66]2) 0    :110101101010: { 1, 3, 5, 6, 8, 10, 0 }
([66]2) -> [+8/-4] 1(X) {6} ([76]1)

{6} ([76]1) 8    :110101010010: { 8, 9, 11, 1, 3, 6 }
```

2. Translate a note to one not in the scale. (8.2, lns. 967 to 1003.)

```
([76]1) -> [+0/-0] 2(7) {6} ([77]4)

{6} ([77]4) 8    :110101001010: { 1, 4, 6, 8, 9, 11 }
```

3. Add a note not in the scale. (8.2, lns. 1005 to 1049.)

```
([77]4) -> [+0/-0] A(1) {7} ([66]4)

{7} ([66]4) 8    :110101101010: { 1, 2, 4, 6, 8, 9, 11 }
```

Even though, a transposition does not alter the harmonic content of a scale, it *does* alter the notes being played. So, it should be counted as a primitive element-by-element scalar operation. Nevertheless, it is because of the very fact that the harmonic content is preserved under transposition that it becomes a factor in our formula for one-note modulations. Also, transpositions are equivalent to a change of key, and to exchange a note for one in the same scale yields the same scale. So, with three primitive operations, we may determine all the one-note modulations between scales in three processes. A one-note modulation is defined in the following formula as a linear superposition of the add and delete functions and a transposition of x semitones:

$$d(\hat{S}_K, m) \oplus a(\hat{S}_K, n) \oplus t(\hat{S}_K, x) = \hat{S}_K - \hat{i}_j m + \hat{i}_j n + x. \tag{1}$$

Let $\hat{S}_K$ be a vector in $\mathbb{Z}_N^{+K}$, for $K \leq N$, and $\hat{i}_j$ be the basis vector of the jth component of $\hat{S}$, for $j \in \mathbb{N}$, $j \leq K$. Let $\hat{Q}_{K-1}$ be a vector in $\mathbb{Z}_N^{+(K-1)}$ for $K-1 \geq 0$, such that $\{s + x | s \in \hat{S}_K\} \setminus \{q \in \hat{Q}_{K-1}\} = \{m + x\}$. We define our delete-note function as

$$d(\hat{S}_K, m) : \{\mathbb{Z}_N^{+K} \times \mathbb{Z}_N \to \mathbb{Z}_N^{+(K-1)} | m \in \hat{S}_K\} = \hat{S}_K - \hat{i}_j m. \tag{2}$$

For $K - 1 \geq 0$, we have no issues. But if otherwise, what are you to subtract from silence? Maybe, we get an inverse modular space or something like that. Our implementation could be extended to allow for such spaces. But we treat silence as absolute and contrary to sound of any kind. Let $\hat{R}_{K+1}$ be a vector of $\mathbb{Z}_N^{+(K+1)}$ as well, for $K + 1 \leq N$, such that $\{r \in \hat{R}_{K+1}\} \setminus \{s \in \hat{S}_K\} = \{n\}$. If $K+1$ is ever greater than N, then whatever pitch is added would be a doubling or an octave of any of the others. Let $\hat{\imath}_j$ be the basis vector of the jth component of $\hat{R}_{K+1}$, for $j \in \mathbb{N}$, $j \leq K+1$. Similarly, we define our add-note function as

$$a(\hat{S}_K, n) : \{\mathbb{Z}_N^{+K} \times \mathbb{Z}_N \to \mathbb{Z}_N^{+(K+1)} | n \in \hat{R}_{K+1}\} = \hat{S}_K + \hat{\imath}_j n. \tag{3}$$

We can also come up with some other neat operations. For example, we can transpose a scale from one key to another by simple modular scalar addition.

$$t(\hat{S}_K, x) : \{\mathbb{Z}_N^{+K} \times \mathbb{Z}_N \to \mathbb{Z}_N^{+K}\} = \hat{S}_K + x. \tag{4}$$

We may change a scale's density while preserving concentration by selecting a scale at a position proportional to the diffusion of first one in their respective densities. In 12-TET, there are only 66 seven-note scales, but 80 six-note scales. A scale with a definition {7}[54] has the same diffusion as {6}[65]. Since

$$54/66 \approx 65/80. \tag{5}$$

3.5 Selecting a Chord

It should be trivial to select a chord by their respective degrees or intervals. We could have an arpeggio stack of thirds of a C major seventh chord in 12-TET in the fourth register: AS({7} ([66]2), (1,4), 2, 3) = 1,5,8,0 = C4,E4,G4,B4.

3.6 How to search a path between modes of any scales?

You know, I wanted to honor previous computer scientist Edger J. Dijkstra by implementing his search algorithm over the graph of all the one-note modulations. Turns out that one: the graph of all the scales in all the keys has 25,344 vertices! Perhaps changing the order in which the graph nodes are listed might yield better results. I will try reverse order, and the order of the ZS1 partition. But having listed them in "Absolute Order", for a simple search between a septatonic scale and a heptatonic scale Dijkstra's Algorithm ran for a about a day and a half. Other options I'm considering are running without the debugger attached, compiling with the Intel **compiler, running on a supercomputer. On the other hand, it also seems fitting to implement a more efficient algorithm like the A*. Mainly, the motivation for this is to measure the accuracy of our theory. Id est, if I can geometrically represent these musical Scale Objects as some vectors in a space, by all means, the shortest path must be a geodesic in such a space. Therefore a geometrically defined geodesic must be the shortest path. QED.

3.7 Generalization

The question remains on whether K note modulations are all linear combinations of one-note modulations. Or if there are L-note modulations that cannot be expressed as a linear combination of K-note modulations for K<L<N. This question could be the first one to solve once the language for music is ready, as a first program.

4 Findings

Rhythm can also be ordered combinatorially with the integer partition function. But that would be the subject for a future article. My integer partition function, although somewhat slow, finds them exactly in the order I need them. I found a Master's thesis by Antoine Zoghbi that showcased an assortment of implementations of the integer partition function and two original ones, the ZS1 and the ZS2 algorithms, for which pseudocode were provided alongside along with their implementations, which the author proves are faster than all the others he knows about. I quickly implemented the ZS1 algorithm and it is fast indeed, compared to mine! But then, I would have to sort them. In all honesty, even if that way was faster in my computer, at least, I think my methods illustrate the idea of the computations being done better than Zoghbi's, whose code for partitioning an integer does not immediately reflect the mental heavy lifting done by means of mathematics.

5 Conclusions

It is sensible to say that we have successfully identified every possible scale for any Equal Temperament Tuning in any natural number of tones. This model presented seems fit enough to serve as a scales engine for a new computer language for Music. Such a language, I would like to implement in Ada.

6 Bibliography

References

[1] Antoine C. Zoghbi. *Algorithms for generating integer partitions.* University of Ottawa, Canada, 1993.

[2] Miller Puckette. *The Theory and Technique of Electronic Music.* World Scientific Publishing Co. Pte. Ltd., 2006.

[3] Rich Cochrane. *Arpeggio and Scale Resources.* Big Noise Publishing, London, UK.

[4] Iannis Xenakis. *Formalized Music.* Pendragon Press, Stuyvesant, NY, 1992.

[5] Arnold Schoenberg. *Theory of Harmony, Translated by Roy E. Carter.* University of California Press, Los Angeles.

[6] Joseph Schillinger. *The Schillinger System of Musical Composition.*

[7] Allen Forte. *The Structure of Atonal Music.* Yale University Press, New Haven and London, 1973.

[8] John Rahn. *Basic Atonal Theory.* Schirmer Books, New York, 1980.

[9] Joseph N. Strauss. *Introduction to Post-Tonal Theory; 4th Edition* W.W. Norton & Company, New York, 2016.

[10] Richard Mark French. *Engineering the Guitar* Springer, New York, 2009.

7 Miscellanea

7.1 Integer Partitions of 12

```
12
1, 11
2, 10
3, 9
4, 8
5, 7
6, 6
1, 1, 10
1, 2, 9
1, 3, 8
1, 4, 7
1, 5, 6
2, 2, 8
2, 3, 7
2, 4, 6
2, 5, 5
3, 3, 6
3, 4, 5
4, 4, 4
1, 1, 1, 9
1, 1, 2, 8
1, 1, 3, 7
1, 1, 4, 6
1, 1, 5, 5
1, 2, 2, 7
1, 2, 3, 6
1, 2, 4, 5
1, 3, 3, 5
1, 3, 4, 4
2, 2, 2, 6
2, 2, 3, 5
2, 2, 4, 4
2, 3, 3, 4
3, 3, 3, 3
1, 1, 1, 1, 8
```

1, 1, 1, 2, 7
1, 1, 1, 3, 6
1, 1, 1, 4, 5
1, 1, 2, 2, 6
1, 1, 2, 3, 5
1, 1, 2, 4, 4
1, 1, 3, 3, 4
1, 2, 2, 2, 5
1, 2, 2, 3, 4
1, 2, 3, 3, 3
2, 2, 2, 2, 4
2, 2, 2, 3, 3
1, 1, 1, 1, 1, 7
1, 1, 1, 1, 2, 6
1, 1, 1, 1, 3, 5
1, 1, 1, 1, 4, 4
1, 1, 1, 2, 2, 5
1, 1, 1, 2, 3, 4
1, 1, 1, 3, 3, 3
1, 1, 2, 2, 2, 4
1, 1, 2, 2, 3, 3
1, 2, 2, 2, 2, 3
2, 2, 2, 2, 2, 2
1, 1, 1, 1, 1, 1, 6
1, 1, 1, 1, 1, 2, 5
1, 1, 1, 1, 1, 3, 4
1, 1, 1, 1, 2, 2, 4
1, 1, 1, 1, 2, 3, 3
1, 1, 1, 2, 2, 2, 3
1, 1, 2, 2, 2, 2, 2
1, 1, 1, 1, 1, 1, 1, 5
1, 1, 1, 1, 1, 1, 2, 4
1, 1, 1, 1, 1, 1, 3, 3
1, 1, 1, 1, 1, 2, 2, 3
1, 1, 1, 1, 2, 2, 2, 2
1, 1, 1, 1, 1, 1, 1, 1, 4
1, 1, 1, 1, 1, 1, 1, 2, 3
1, 1, 1, 1, 1, 1, 2, 2, 2
1, 1, 1, 1, 1, 1, 1, 1, 1, 3
1, 1, 1, 1, 1, 1, 1, 1, 2, 2
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1

7.2 The Man Machine

[1] :000000000000:

[1] :100000000000:

[1] :110000000000:
[2] :101000000000:
[3] :100100000000:
[4] :100010000000:
[5] :100001000000:
[6] :100000100000:

[1] :111000000000:
[2] :110100000000:
[3] :110000000010:
[4] :110010000000:
[5] :110000000100:
[6] :110001000000:
[7] :110000001000:
[8] :110000100000:
[9] :110000010000:
[10] :101010000000:
[11] :101001000000:
[12] :101000000100:
[13] :101000100000:
[14] :101000001000:
[15] :101000010000:
[16] :100100100000:
[17] :100100010000:
[18] :100100001000:
[19] :100010001000:

[1] :111100000000:
[2] :111010000000:
[3] :111000000010:
[4] :110110000000:
[5] :111001000000:
[6] :111000000100:
[7] :110011000000:
[8] :111000100000:
[9] :111000001000:
[10] :110001100000:
[11] :111000010000:
[12] :110000110000:
[13] :110101000000:
[14] :110100000010:
[15] :110000001010:
[16] :110100100000:
[17] :110100000100:
[18] :110010100000:
[19] :110010000010:
[20] :110000010100:
[21] :110000010010:
[22] :110100010000:
[23] :110100001000:
[24] :110001010000:
[25] :110001000010:
[26] :110000101000:
[27] :110000100010:
[28] :110010010000:
[29] :110010000100:
[30] :110000100100:
[31] :110010001000:
[32] :110001001000:
[33] :110001000100:
[34] :101010100000:
[35] :101010010000:
[36] :101010000100:
[37] :101001010000:
[38] :101010001000:
[39] :101000101000:
[40] :101001001000:
[41] :101001000100:
[42] :101000100100:
[43] :100100100100:

[1] :111110000000:
[2] :111101000000:
[3] :111100000010:
[4] :111011000000:
[5] :111000000110:
[6] :111100100000:
[7] :111100000100:
[8] :111001100000:
[9] :111000001100:
[10] :111100010000:
[11] :111100001000:
[12] :111000110000:
[13] :111000011000:
[14] :111010100000:
[15] :111010000010:
[16] :111000001010:
[17] :110110100000:
[18] :110110000010:
[19] :110101100000:
[20] :111010010000:
[21] :111010000100:
[22] :111001010000:
[23] :111001000010:
[24] :111000010100:
[25] :111000010010:
[26] :110110010000:
[27] :110110000100:
[28] :110100110000:
[29] :110100001100:
[30] :110011000010:
[31] :110010110000:
[32] :111010001000:
[33] :111000101000:
[34] :111000100010:
[35] :110110001000:
[36] :110100011000:
[37] :110001100010:
[38] :111001001000:
[39] :111001000100:
[40] :111000100100:
[41] :110011001000:
[42] :110011000100:
[43] :110010011000:
[44] :110101010000:
[45] :110101000010:
[46] :110100001010:
[47] :110000101010:
[48] :110101001000:
[49] :110101000100:
[50] :110100101000:
[51] :110100100010:
[52] :110100010100:
[53] :110100010010:
[54] :110010101000:
[55] :110010100010:
[56] :110010001010:
[57] :110001010100:
[58] :110001010010:
[59] :110001001010:
[60] :110100100100:
[61] :110010100100:
[62] :110010010100:
[63] :110010010010:
[64] :101010101000:
[65] :101010100100:
[66] :101010010100:

[1] :111111000000:
[2] :111110100000:
[3] :111110000010:
[4] :111101100000:
[5] :111100000110:
[6] :111011100000:
[7] :111110010000:
[8] :111110000100:
[9] :111100110000:
[10] :111100001100:
[11] :111001110000:
[12] :111110001000:
[13] :111100011000:
[14] :111000111000:
[15] :111101010000:
[16] :111101000010:
[17] :111100001010:
[18] :111011010000:
[19] :111011000010:
[20] :111010110000:
[21] :111010000110:
[22] :111000011010:
[23] :111000010110:
[24] :110110110000:
[25] :111101001000:
[26] :111101000100:
[27] :111100101000:
[28] :111100100010:
[29] :111100010100:
[30] :111100010010:
[31] :111011001000:
[32] :111011000100:
[33] :111010011000:
[34] :111010001100:
[35] :111001101000:
[36] :111001100010:
[37] :111001011000:
[38] :111001000110:
[39] :111000110100:
[40] :111000110010:
[41] :111000101100:
[42] :111000100110:
[43] :110110011000:
[44] :110110001100:
[45] :111100100100:
[46] :111001100100:
[47] :111001001100:
[48] :110011001100:
[49] :111010101000:
[50] :111010100010:
[51] :111010001010:
[52] :111000101010:
[53] :110110101000:
[54] :110110100010:
[55] :110110001010:
[56] :110101101000:
[57] :110101100010:
[58] :110101011000:
[59] :111010100100:
[60] :111010010100:
[61] :111010010010:
[62] :111001010100:

```
[63]  :111001010010:
[64]  :111001001010:
[65]  :110110100100:
[66]  :110110010100:
[67]  :110110010010:
[68]  :110101100100:
[69]  :110101001100:
[70]  :110100110100:
[71]  :110100110010:
[72]  :110100101100:
[73]  :110011001010:
[74]  :110010110010:
[75]  :110101010100:
[76]  :110101010010:
[77]  :110101001010:
[78]  :110100101010:
[79]  :110010101010:
[80]  :101010101010:

[1]  :111111100000:
[2]  :111111010000:
[3]  :111111000010:
[4]  :111110110000:
[5]  :111110000110:
[6]  :111101110000:
[7]  :111100001110:
[8]  :111111001000:
[9]  :111111000100:
[10]  :111110011000:
[11]  :111110001100:
[12]  :111100111000:
[13]  :111100011100:
[14]  :111110101000:
[15]  :111110100010:
[16]  :111110001010:
[17]  :111101101000:
[18]  :111101100010:
[19]  :111101011000:
[20]  :111101000110:
[21]  :111100011010:
[22]  :111100010110:
[23]  :111011101000:
[24]  :111011100010:
[25]  :111011011000:
[26]  :111011000110:
[27]  :111010111000:
[28]  :111000110110:
[29]  :111110100100:
[30]  :111110010100:
[31]  :111110010010:
[32]  :111101100100:
[33]  :111101001100:
[34]  :111100110100:
[35]  :111100110010:
[36]  :111100101100:
[37]  :111100100110:
[38]  :111011100100:
[39]  :111011001100:
[40]  :111010011100:
[41]  :111001110010:
[42]  :111001101100:
[43]  :111001100110:
[44]  :111101010100:
[45]  :111101010010:
[46]  :111101001010:
[47]  :111100101010:
[48]  :111011010100:
[49]  :111011010010:
[50]  :111011001010:
[51]  :111010110100:
[52]  :111010110010:
[53]  :111010101100:
[54]  :111010100110:
[55]  :111010011010:
[56]  :111010010110:
[57]  :111001101010:
[58]  :111001011010:
[59]  :111001010110:
[60]  :110110110100:
[61]  :110110110010:
[62]  :110110101100:
[63]  :110110011010:
[64]  :111010101010:
[65]  :110110101010:
[66]  :110101101010:

[1]  :111111110000:
[2]  :111111101000:
[3]  :111111100010:
[4]  :111111011000:
[5]  :111111000110:
[6]  :111110111000:
[7]  :111110001110:
[8]  :111101111000:
[9]  :111111100100:
[10]  :111111001100:
[11]  :111110011100:
[12]  :111100111100:
[13]  :111111010100:
[14]  :111111010010:
[15]  :111111001010:
[16]  :111110110100:
[17]  :111110110010:
[18]  :111110101100:
[19]  :111110100110:
[20]  :111110011010:
[21]  :111110010110:
[22]  :111101110100:
[23]  :111101110010:
[24]  :111101101100:
[25]  :111101100110:
[26]  :111101011100:
[27]  :111101001110:
[28]  :111100111010:
[29]  :111100110110:
[30]  :111100101110:
[31]  :111011101100:
[32]  :111011100110:
[33]  :111011011100:
[34]  :111110101010:
[35]  :111101101010:
[36]  :111101011010:
[37]  :111101010110:
[38]  :111011101010:
[39]  :111011011010:
[40]  :111011010110:
[41]  :111010111010:
[42]  :111010110110:
[43]  :110110110110:

[1]  :111111111000:
[2]  :111111110100:
[3]  :111111110010:
[4]  :111111101100:
[5]  :111111100110:
[6]  :111111011100:
[7]  :111111001110:
[8]  :111110111100:
[9]  :111110011110:
[10]  :111111101010:
[11]  :111111011010:
[12]  :111111010110:
[13]  :111110111010:
[14]  :111110110110:
[15]  :111110101110:
[16]  :111101111010:
[17]  :111101110110:
[18]  :111101101110:
[19]  :111011101110:

[1]  :111111111100:
[2]  :111111111010:
[3]  :111111110110:
[4]  :111111101110:
[5]  :111111011110:
[6]  :111110111110:

[1]  :111111111110:

[1]  :111111111111:

The Man Machine
(c) 2025 Reed A.
(c) 2025 G. Davila
```

7.3 Source Code License

```
The Source Made Available (SMA) License for Personal
    Reference or Publication Review (PRoPR).
The SMA-PRoPR License.
Copyright 2025 Axewielder Goods
Copyright 2025 Guillermo M. Davila
```

8 Code

8.1 IntPart.h

```
/******************************************************************************
*   IntPart.h                                                                 *
*******************************************************************************
*   2025 (c) Guillermo M. Davila Andino                                       *
*   All rights reserved.                                                      *
*******************************************************************************
```

```
*    This header file implements an integer partition function "partition"
*
*    to part unsigned integers into k parts in standard ISO C++20.
*
******************************************************************************/

#pragma once
#include <vector>
#include <string>
#include <iostream>
#include <set>

void PrntPrts(std::vector<unsigned long long> Parts) {
    std::string Partition;
    for (unsigned long long j = 0; j < Parts.size(); j++) {
        Partition = Partition + std::to_string(Parts[j]);
        if (j >= 0 && (j + 1 < Parts.size())) Partition = Partition + ", ";
    }
    std::cout << Partition << std::endl;
}

std::set<std::vector<unsigned long long>>
partition(unsigned long long Number, unsigned long long Parts) {
    unsigned long long Sum = 0;
    std::vector<unsigned long long> Par;

    std::set<std::vector<unsigned long long>> PartitionTree;
    std::set<std::vector<unsigned long long>> Partitions;

    if (Parts == 0) {
        if (Number == 0) {
            Par.push_back(0);
            Partitions.insert(Par);
        }
        return Partitions;
    }
    for (unsigned long long i = 0; i < Parts; i++) {
        Par.push_back(1);
        Sum++;
    }

    while (Sum <= Number) {
        if (Sum == Number) {
            Partitions.insert(Par);
        }
        else PartitionTree.insert(Par);
        if (Par.size() > 0) Par[Parts - 1]++;
        Sum = 0;
        for (unsigned long long i : Par) {
            Sum += i;
        }
    }

    for (unsigned long long i = 0; i < Number; i++) {
        while (!PartitionTree.empty()) {
            Par = *PartitionTree.begin();
            PartitionTree.erase(PartitionTree.begin());

            if (Par[Parts - 1] >= i) {
                for (unsigned long long k = 1; k <= Parts; k++) {
                    Par[Parts - k]++;
                    Sum = 0;
                    for (unsigned long long l : Par) Sum += l;

                    if (Sum == Number) {
                        Partitions.insert(Par);
                    }
                    if (Par[Parts - 1] >= i + 1 && Sum < Number) {
```

```
                            PartitionTree.insert(Par);
                        }
                    }
                }
            }
    }
    return Partitions;
}

std::set<std::vector<unsigned long long>> IntPart(unsigned long long Number) {
    unsigned long long Sum = 0;
    std::vector<unsigned long long> Par;

    std::set<std::vector<unsigned long long>> PartitionTree;
    std::set<std::vector<unsigned long long>> NewPartTree;
    std::set<std::vector<unsigned long long>> Partitions;


    if (Number == 0) {
        Par.push_back(0);
        Partitions.insert(Par);
        return Partitions;
    }

    for (unsigned long long u = 1; u <= Number; u++) {
        Par.push_back(1);
        PartitionTree.insert(Par);
    }
    //Sum = 1;



    while (!PartitionTree.empty()) {
        Par = *PartitionTree.begin();


        PartitionTree.erase(PartitionTree.begin());
        /*Sum = 0;
        for (unsigned long long l : Par) Sum += l;*/
        if (Sum == Number) {
            //std::sort(Par.begin(), Par.end());
            Partitions.insert(Par);
            Sum = (*PartitionTree.begin())[0];

        }
        else if (Sum < Number) {
            for (unsigned long long k = 1; k <= Par.size(); k++) {
                Par[Par.size() - k]++;
                NewPartTree.insert(Par);
                //Sum++;
            }
            Sum++;
        }



    }

    return Partitions;
}


// This is a quick and dirty translation of the code I found in a paper... not my idea
std::vector<std::vector<unsigned long long>> ZS1(unsigned long long n) {
    unsigned long long t;
    unsigned long long r;

    std::vector<unsigned long long> x;
```

```
    std::vector<std::vector<unsigned long long>> Partitions;

    x.push_back(n);
    Partitions.push_back(x);
    x.pop_back();
    for (unsigned long long i = 1; i <= n; i++) {
        x.push_back(1);
    }
    x[0] = n;
    unsigned long long m = 1;
    unsigned long long h = 1;

    while (x[0] != 1) {
        if (x[h - 1] == 2) {
            m++;
            x[h - 1] = 1;
            h--;
        }
        else {
            r = x[h - 1] - 1;
            t = m - h + 1;
            x[h - 1] = r;
            while (t >= r) {
                h++;
                x[h - 1] = r;
                t = t - r;
            }
            if (t == 0) m = h;
            else {
                m = h + 1;
                if (t > 1) {
                    h++;
                    x[h - 1] = t;
                }
            }
        }
        Partitions.push_back(x);
    }
    return Partitions;
}
```

8.2 Scales2B.h

```
// (c) Reed Axewielder 2024 - 2025 // GMDA

#pragma once

#include <algorithm>
#include <iostream>
#include <fstream>
#include <numbers>
#include <cmath>
#include <map>
#include <set>
#include <string>
#include <vector>
#include <filesystem>
#include <queue>
#include <chrono>
#include "IntPart.h"

std::string PitchName(unsigned P) {
    return std::to_string(P);
}

typedef std::pair<unsigned, unsigned> iPitch;

```

```
const unsigned SamplingRate = 96000;

class ScaleDef
{
private:
    std::string Name;
    std::vector <std::string> Modes;
    // (Note, Degree)
    std::map <unsigned, unsigned> iNotes;
    std::map <unsigned, unsigned>::size_type iDegs = 0;
    unsigned Tones = 0;

public:

    ScaleDef() {
        iDegs = 0;
        Name.clear();
        Modes.clear();
        iNotes.clear();
        Tones = 0;
    }
    ScaleDef(std::string ScaleName, std::map <unsigned, unsigned>::size_type Degrees,
        std::set <unsigned> DefNotes, std::vector <std::string> ModeNames, unsigned TonesNumber)
    {
        Name = ScaleName;
        if (ModeNames.size() == Degrees) Modes = ModeNames;
        else std::cerr << "Invalid amount of Mode Names";
        Tones = TonesNumber;
        iDegs = Degrees;
        if (DefNotes.size() == iDegs) for (unsigned i : DefNotes) insert(i);
        else std::cerr << "Scale Definition Invalid." << std::endl;
    }
    // iDegs is the same as the number of notes in the Scale.
    std::map <unsigned, unsigned>::size_type GetDegs() {
        return iDegs;
    }
    unsigned GetTones() {
        return Tones;
    }
    void SetTones(unsigned tones) {
        Tones = tones;
    }
    std::string PrintDef() {
        std::string Definition;
        for (unsigned P = 0; P < Tones; P++) {
            if (iNotes.contains(P)) {
                Definition = Definition + "1";
            }
            else {
                Definition = Definition + "0";
            }
        }
        return Definition;
    }
    std::string PrintModeDef(unsigned m) {
        std::string buffer = PrintDef();
        std::rotate(buffer.begin(), buffer.begin() + GetNote(m), buffer.end());
        return buffer;

    }
    void SetNotes(std::set <unsigned> DefNotes) {
        iNotes.clear();
        for (unsigned i : DefNotes) insert(i);

    }

    // Returns the note of the ScaleDef at the designed degree.
    unsigned GetNote(unsigned int Degree) {
```

```
        if (Degree < iDegs) {
            for (iPitch Note : iNotes) {
                if (get<1>(Note) == Degree) {
                    return get<0>(Note);
                }
            }
        }
        else std::cerr << "Degree out of bounds.";
        return B;
    }
    void SetNotes(std::map <unsigned, unsigned> DefNotes, unsigned Tones) {
        iNotes.clear();
        iNotes = DefNotes;
        iDegs = iNotes.size();
        ScaleDef::Tones = Tones;
    }
    void SetDef(ScaleDef Def) {
        Name = Def.Name;
        Modes = Def.Modes;
        iDegs = Def.iDegs;
        iNotes = Def.iNotes;
        Tones = Def.Tones;
    }
    std::string GetName() {
        return Name;
    }
    // Here, we have our tit for one note operations.
    bool insert(unsigned p) {
        if (p < Tones) {
            if (!iNotes.contains(p)) {

                std::set<unsigned> scaledegs;

                for (auto& i : iNotes) {
                    scaledegs.insert(i.first);
                }
                iNotes.clear();
                scaledegs.insert(p);
                unsigned count = 0;

                for (unsigned u : scaledegs) {
                    iNotes.insert(std::make_pair(u, count));
                    count++;
                }
                iDegs = iNotes.size();

                return true;
            }
            else return false;
        }
        else return false;
    }


    // Here, is the other one, for one note operations.
    bool remove(unsigned p) {
        if (p < Tones) {
            if (iNotes.contains(p)) {
                iNotes.erase(iNotes.find(p));
                std::set<unsigned> scaledegs;

                for (auto& i : iNotes) {

                    scaledegs.insert(i.first);
                }

                iNotes.clear();
                unsigned count = 0;
```

```
                for (unsigned u : scaledegs) {
                    iNotes[u] = count;
                    count++;
                }
                iDegs = iNotes.size();
                return true;
            }
            else {
                return false;
            }
        }
        else return false;
    }









    void SetName(std::string N) {
        Name = N;
    }

    void PushMode(std::string M) {
        Modes.push_back(M);
    }

    // Returns Mode name of degree i
    std::string GetMode(unsigned i) {
        return Modes[i];
    }

    // Returns the Mode name of degree of Note
    std::string GetModeOf(unsigned Note) {
        auto deg = GetDeg(Note);
        if (deg != -1) {
            return GetMode(deg);
        }
        std::cerr << PitchName(Note) << " Not in scale." << std::endl;
        return "NaP";
    }

    std::map <unsigned, unsigned> GetNotes() const {
        return iNotes;
    }

    ScaleDef GetDef() {
        return ScaleDef(*this);
    }
    // Returns the degree of the pitch P in the ScaleDef.
    int GetDeg(unsigned P) {
        if (iNotes.contains(P)) {
            return iNotes[P];
        }

        std::cerr << "Pitch not in scale." << std::endl;
        return -1;
    }
    std::map <unsigned, unsigned> Difference(std::map <unsigned, unsigned> PitchB) {
        std::map <unsigned, unsigned> V;

        std::set_difference(
            iNotes.begin(), iNotes.end(),
            PitchB.begin(), PitchB.end(),
            std::inserter(V, V.begin()),
```

```
            [](const auto& a, const auto& b) { return a.first < b.first; }
        ); // XD
        return V;
    }


    // Here, M is the degree of the mode. M is in [0, iDegs).
    // This function returns an indexed map of the notes of the mode
    // with root in 0.
    std::map <unsigned, unsigned> Mode(unsigned M) {
        std::map <unsigned, unsigned> mode;
        if (M < iDegs) {
            unsigned Root = GetNote(M);

            for (unsigned int i = 0; i < iDegs; i++) {
                unsigned Note = GetNote((static_cast<unsigned long long>(M) + i) % iDegs);

                // Here, the pitch is mapped to the degree of the mode,
                // not to the mode names.
                // Otherwise, our line would have been this:
                // mode[(unsigned)((Tones + (int)Note - (int)Root) % Tones)] = (M + i) % iDegs;
                mode[(unsigned)((Tones + (int)Note - (int)Root) % Tones)] = i;
            }
        }
        return mode;
    }

};

class Scale : public ScaleDef
{
private:
    unsigned Tonic = B;
    unsigned iMode = 0;
    std::set <unsigned> iScale;
    std::filesystem::path ScalePath;

public:
    Scale(unsigned Key, ScaleDef Def) {
        Tonic = Key;
        SetDef(Def);
        Modify(0);
        ChangeKey(Key);
    }
    Scale() {
        Tonic = B;
    }
    // Use this function every time, to initialize our Pitch vector.
    //      Scale[i].Modify(0);
    // Use after SetNotes function. Use before ChangeKey function.
    // M is the scale degree. M is in [0, iDegs).
    // This function fills the iScale with the notes of the scale
    // in mode M with root in Tonic.
    unsigned Modify(unsigned M) {
        unsigned Root = 0;
        if (GetDegs() > 0) {
            Root = unsigned((ScaleDef::GetTones() + unsigned(GetNote(M)) + unsigned(Tonic))
                % ScaleDef::GetTones());
            std::map <unsigned, unsigned> Buffer;
            if (M < GetDegs()) {
                iMode = M;
                Buffer = Mode(M);
                iScale.clear();
                for (const auto& P : Buffer) {
                    iScale.insert(
                        unsigned((ScaleDef::GetTones() +
                        unsigned((unsigned)P.first) +
                        unsigned(Tonic)) %
```

```
                        ScaleDef::GetTones()));
                }
            }
            return Root;
        }
        else return 0;

    }
    Scale GetScale() { return Scale(*this); }
    void SetPath(std::filesystem::path P) {
        ScalePath = P;
    }
    std::filesystem::path GetPath() {
        return ScalePath;
    }
    unsigned GetKey() const { return Tonic; }
    void ChangeKey(unsigned Key) {
        unsigned OldKey = Tonic;
        auto OldScale = GetiScale();
        unsigned Dif = (GetTones() + Key - Tonic) % GetTones();
        Tonic = Key;
        iScale.clear();
        for (const auto& P : OldScale) {
            iScale.insert((P + Dif) % GetTones());
        }
    }
    std::set <unsigned> GetiScale() { return iScale; }
    unsigned int GetiMode() {
        return iMode;
    }
    std::string ModeName() {
        return GetMode(GetiMode());
    }
    std::set <unsigned> SetDifference(std::set <unsigned> PitchB) {
        std::set <unsigned> V;
        // std::set_difference return all elements in the first one not in the second.
        std::set_difference(
            iScale.begin(), iScale.end(),
            PitchB.begin(), PitchB.end(),
            std::inserter(V, V.begin())
        ); // XD
        return V;
    }
    void transpose(int Steps) {
        unsigned transposition = (GetKey() + Steps) % GetTones();
        ChangeKey(transposition);
    }

    bool DeleteNote(unsigned N) {
        SetNotes(Mode(iMode), GetTones());
        unsigned DefNote = (GetTones() + N - Tonic) % GetTones();
        if (iScale.contains(N)) {

            remove(DefNote);

            return true;
        }
        return false;

    }

    bool AddNote(unsigned N) {
        unsigned DefNote = (GetTones() + N + GetNote(iMode) - Tonic) % GetTones();
        if (!iScale.contains(N)) {

            insert(DefNote);

            return true;
```

```
        }
        return false;
    }
    // Returns the degree of a Pitch in the mode of the iScale.
    unsigned GetiDeg(unsigned Pitch) {
        unsigned p = (GetTones() + Pitch + GetNote(iMode) - Tonic) % GetTones();
        return (GetDegs() + GetDeg(p) - iMode) % GetDegs();
    }
};

unsigned int weight(std::string chars, Scale* Candidate) {
    for (unsigned int i = 0; i < chars.size(); i++) {
        if (chars[i] == '1') {

            (*Candidate).insert(i);
        }
    }
    return (*Candidate).GetDegs();
}


// this is a mode test
static bool mode2(std::string known, std::string candidate)
{
    bool mode = false;

    std::string buffer = candidate;
    for (unsigned int i = 0; i < known.size(); i++) {
        if (buffer == known) {
            mode = true;
            break;
        }
        std::rotate(buffer.begin(), buffer.begin() + 1, buffer.end());
    }

    return mode;
}

int modedist(std::string known, std::string candidate)
{
    unsigned dist = -1;

    std::string buffer = candidate;
    for (unsigned int i = 0; i < known.size(); i++) {
        if (buffer == known) {
            dist = i;
            break;
        }
        // this is a left-rotation by 1 element. (buffer << 1)
        std::rotate(buffer.begin(), buffer.begin() + 1, buffer.end());
    }

    return dist;
}


bool known(std::vector<Scale> scales, ScaleDef candidate) {
    bool known = false;
    if (scales.size() == 0) return false;
    for (Scale i : scales) {
        known = known || mode2(i.PrintDef(), candidate.PrintDef());
    }
    return known;
}

bool known2(std::vector<std::string> scales, std::string candidate) {
    bool known = false;
    if (scales.size() == 0) return false;
```

```
    for (std::string i : scales) {
        known = known || mode2(i, candidate);
    }
    return known;
}

// Returns the Scale Number [0,N) and the Transposition needed
std::pair<int, int> knownu(std::vector<Scale> scales, ScaleDef candidate) {
    if (scales.size() == 0) return std::make_pair(-1, -1);
    for (unsigned u = 0; u < scales.size(); u++) {
        int dist = modedist(scales[u].PrintDef(), candidate.PrintDef());
        if (dist >= 0) {
            return std::make_pair(u, unsigned(dist));
        }
    }

    return std::make_pair(-1, -1);
}


// This function takes in std::string to binary representation
std::string tobin(std::string s) {
    std::string buffer;
    for (char c : s) {
        buffer = buffer + "1";
        for (unsigned i = 1; i < c; i++) {
            buffer = buffer + "0";
        }
    }
    return buffer;
}

// scalet = (NoteOperation, Transposition)
typedef std::pair < std::string, unsigned> scalet;
// AtlasKey = (notes, scale, mode)
typedef std::tuple<unsigned, unsigned, unsigned> AtlasKey;
// AtlasVal = (Next, (NoteOperation, Transposition))
typedef std::pair <std::string, scalet> AtlasVal;

// graph Graph[(notes, scale, mode)] = [(Next, (NoteOperation, Transposition))];
typedef std::map < AtlasKey, std::vector<AtlasVal> > graph;


// graph2 DegMods[(ScaleName, Transposition)] = [(Next, Operation)];
typedef std::map < std::pair <std::string, int>,
    std::vector <std::pair <std::string, std::string> > > graph2;



std::string encode(unsigned Notes, unsigned Scale, unsigned Mode) {
    return "{" + std::to_string(Notes) + "}-([" + std::to_string(Scale) + "]"
        + std::to_string(Mode) + ")";
}

std::string EncodeKey(AtlasKey MyKey) {
    return encode(get<0>(MyKey), get<1>(MyKey), get<2>(MyKey));
}


// (Notes, Scale, Mode)
AtlasKey decode(std::string state) {
    std::string buf;
    unsigned scale;
    unsigned mode;
    unsigned notes = 0;

    unsigned ind = 1;
    if (state[0] == '{') {
```

```
        buf.clear();
        while (state[ind] != '}') {
            buf = buf + state[ind];
            ind++;
        }
        notes = std::stoi(buf);
        while (state[ind] != '[') {
            ind++;
        }
        ind++;

    }
    buf.clear();
    for (unsigned i = ind; i < state.size(); i++) {

        if (state[i] != ']') {
            buf = buf + state[i];
        }
        else {
            scale = stoi(buf);
            buf.clear();
            i++;
            while (state[i] != ')') {
                buf = buf + state[i];
                i++;
            }
            mode = stoi(buf);
            break;
        }
    }
    return std::make_tuple(notes, scale, mode);
}
// (Notes-1, Scale-1, Mode-1)
AtlasKey decode2(std::string state) {
    std::string buf;
    unsigned scale;
    unsigned mode;
    unsigned notes = 0;

    unsigned ind = 1;
    if (state[0] == '{') {
        buf.clear();
        while (state[ind] != '}') {
            buf = buf + state[ind];
            ind++;
        }
        notes = std::stoi(buf) - 1;
        while (state[ind] != '[') {
            ind++;
        }
        ind++;

    }
    buf.clear();
    for (unsigned i = ind; i < state.size(); i++) {

        if (state[i] != ']') {
            buf = buf + state[i];
        }
        else {
            scale = stoi(buf) - 1;
            buf.clear();
            i++;
            while (state[i] != ')') {
                buf = buf + state[i];
                i++;
            }
            mode = stoi(buf) - 1;
```

```
            break;
        }
    }
    return std::make_tuple(notes, scale, mode);
}

// 3. Get the next steps.
std::vector<std::pair<AtlasKey, unsigned>>
next(graph G, AtlasKey ScaleT, unsigned Key, unsigned Tones) {
    std::vector<std::pair<AtlasKey, unsigned>> n;

    std::vector<AtlasVal> NXT = G[ScaleT];

    for (AtlasVal i : NXT) {
        auto nxtTup = decode(i.first);// std::make_tuple(i.first, i.second, transposition);
        auto nxtKey = (Key + i.second.second) % Tones;
        auto nxt = std::make_pair(nxtTup, nxtKey);
        n.push_back(nxt);
    }
    return n;
}

// 4. Get the previous steps.
std::vector<std::pair<AtlasKey, scalet>>
prev(graph G, AtlasKey ScaleT, unsigned Tones) {
    std::vector<std::pair<AtlasKey, scalet>> previous;

    for (auto& Rule : G) {

        for (auto& nextone : Rule.second) {
            if (decode(nextone.first) == ScaleT) {
                previous.push_back(std::make_pair(Rule.first, nextone.second));
            }
        }

    }
    return previous;
}

static double VecDist(std::vector<unsigned> a, std::vector<unsigned> b) {
    size_t maxSize = std::max(a.size(), b.size());
    double sum = 0.0;

    for (size_t i = 0; i < maxSize; i++) {
        unsigned long long ai = (i < a.size()) ? static_cast<unsigned long long>(a[i]) : 0;
        unsigned long long bi = (i < b.size()) ? static_cast<unsigned long long>(b[i]) : 0;
        sum += (ai > bi ? ai - bi : bi - ai) * (ai > bi ? ai - bi : bi - ai);
    }

    return std::sqrt(sum);
}


class Sound {
public:
    std::map<long long, std::vector<float>> SamplesList;
    Sound() = default;
};


class Tuning {
public:

    std::map<long long, double> PitchCollection;

    Tuning() = default;
};

```

```
class EqualTemperament : public Tuning {
public:
    std::vector <std::vector <std::string>> ScaleDefVector;
    std::vector<std::filesystem::path> ScaleListVector;
    std::vector <std::vector <Scale>> ScalesVector;
    graph ModAtlas;
    std::set<unsigned> Gamut;

    double ReferencePitch;      // in Hz
    unsigned ReferenceTone;
    int ReferenceOctave;
    unsigned Tones;   // number of tones per octave
    long long ReferenceNoteNumber;


    long long NoteNumber(unsigned pitch, int octave) {
        return 40 + Tones * long long(octave - 4) + long long(pitch - 1);
    }

    double NoteFrequency(unsigned pitch, int octave) {
        return ReferencePitch * std::pow(double(2), double(double(NoteNumber(pitch, octave)
            - NoteNumber(ReferenceTone, ReferenceOctave)) / Tones));
    }
    double NoteFrequency(long long Number) {
        return ReferencePitch * std::pow(double(2),
            double(double(Number - NoteNumber(ReferenceTone, ReferenceOctave)) / Tones));
    }

    EqualTemperament(unsigned long long T, unsigned RT, int RO, double RP) {
        ReferencePitch = RP;
        ReferenceOctave = RO;
        ReferenceTone = RT;
        Tones = T;
        ReferenceNoteNumber = NoteNumber(ReferenceTone, ReferenceOctave);
        for (int octave = -64; octave <= 64; octave++) {
            for (unsigned pitch = Tones; pitch > 0; pitch--) {
                PitchCollection[ReferenceNoteNumber] =
                    NoteFrequency(ReferenceTone, ReferenceOctave);

            }
        }

        Start2();

    }
    EqualTemperament() = default;

    int ConstructScales() {

        ScaleDefVector.clear();

        std::vector<std::vector<std::string>> ScalePartitionsVector;
        std::vector<std::string> ScalePartitions;
        std::set<std::vector<unsigned long long>> KParts;
        std::string part;

        for (unsigned long long i = 1; i <= Tones; i++) {
            KParts.clear();
            KParts = partition(Tones, i);
            ScalePartitions.clear();

            for (std::vector<unsigned long long> j : KParts) {
                part.clear();
                for (unsigned long long k : j) {
                    part = part + char(k);
                }

                ScalePartitions.push_back(part);
```

```
            }
            ScalePartitionsVector.push_back(ScalePartitions);
        }

        std::vector <std::string> Buffer;
        std::vector <std::vector<std::string>> BufferVector;
        for (std::vector <std::string> i : ScalePartitionsVector) {
            for (std::string j : i) {
                std::string j0 = j;
                do {
                    if (!known2(Buffer, j0)) {
                        Buffer.push_back(j0);
                        //for (char k : j0) {
                        //   std::cout << std::to_string(int(k)) << " ";}
                    }
                } while (std::next_permutation(j0.begin(), j0.end()));
            }
            ScaleDefVector.push_back(Buffer);
            Buffer.clear();
        }

        unsigned long long Sum = 0;
        for (auto& i : ScaleDefVector) {
            Sum += i.size();
            std::cout << i.size() << std::endl;
        }

        std::cout << "\nTotal of " << Sum << " scales in " << Tones << " tones.\n";

        return 0;
    }

    int GenerateScaleList() {
        //*****************************************
        // This routine generates the scale list that then can be used to
        // find their distances, whether by one-note transformation, or by
        // the eucludean method.
        // This must only happen after the scales have been computed.
        // That is, after the previous routine.

        std::string ScaleList0 = ".\\" + std::to_string(Tones) + "Tones\\";
        std::string folder = "md " + ScaleList0;
        system(folder.c_str());
        std::fstream file;

        for (auto& i : ScaleDefVector) {
            ScaleList0 = ".\\" + std::to_string(Tones)
                + "Tones\\SL" + std::to_string(i[0].length())
                + "Notes" + std::to_string(Tones) + "Tones.txt";
            ScaleListVector.push_back(ScaleList0);
            file.clear();
            file.open(ScaleList0.c_str(), std::fstream::out);
            unsigned int count = 0;
            for (std::string s : i) {
                if (count > 0) file << std::endl;
                file << "[" << count + 1 << "]:" << tobin(s) << ":";
                for (unsigned j = 0; j < s.length(); j++) {
                    file << "([" << count + 1 << "]" << j + 1 << ")";
                    if (j != s.length() - 1) file << ",";
                }
                count++;
            }
            file.close();
        }

        int Sum = 0;
        for (auto& i : ScaleListVector) {
            if (std::filesystem::exists(i)) Sum += 0;
```

```
            else {
                std::cerr << "No Scale List exists." << std::endl;
                Sum += -1;
            }
        }
        return Sum;

    }

    // 3. Create the Scale Objects
    int CreateScales() {
        std::cout << "ScaleListVector size= " << ScaleListVector.size() << std::endl;
        for (unsigned long long i = 0; i < ScaleListVector.size(); i++) {

            // Create all scales
            std::string name;
            std::string notes;
            std::string mode;

            Scale candidate;

            std::vector <Scale> Scales;
            Scales.clear();

            std::fstream file;
            file.open(ScaleListVector[i].c_str());
            while (file.good()) {

                name.clear();
                candidate = Scale();
                candidate.SetTones(Tones);
                std::getline(file, name, ':');
                candidate.SetName(name);
                notes.clear();
                std::getline(file, notes, ':');
                if (weight(notes, &candidate) == i + 1) {

                    if (!known(Scales, candidate)) {

                        for (int j = 0; j < i; j++) {
                            mode.clear();
                            getline(file, mode, ',');
                            candidate.PushMode(mode);
                        }
                        mode.clear();
                        getline(file, mode);
                        candidate.PushMode(mode);

                        candidate.SetPath(ScaleListVector[i].c_str());
                        Scales.push_back(candidate);
                    }
                }
            }
            file.close();
            ScalesVector.push_back(Scales);

        }

        return 0;
    }

    // 4. Compute the Modulation Maps.


    // 4.2. Compute Key Modulations.
    int ComputeKeyModulations() {
        /**********************************************************************
        *   Here, we compute the Modulations between keys and scales.
```

```
*/


int Resolution = 0;

std::string name;
std::string notes;
std::string mode;
std::string filename;

Scale ModeA;
Scale ModeB;
std::fstream file;

std::string folder = "md .\\" + std::to_string(Tones) + "Tones\\KeyMods\\";
system(folder.c_str());



Scale Chromatic;
Chromatic.SetTones(Tones);

for (unsigned u = 0; u < Tones; u++) {
    Chromatic.insert(u);
}
Chromatic.Modify(0);
std::cout << "chromatic size: " << Chromatic.GetDegs() << std::endl;

std::cout << ScalesVector.size() << std::endl;
for (unsigned long long u = 0; u < ScalesVector.size(); u++) {

    std::cout << ScalesVector[u].size() << std::endl;
    for (unsigned v = 0; v < ScalesVector[u].size(); v++) {
        name.clear();
        name = ScalesVector[u][v].GetName();
        size_t found = name.find(" ");
        while (found != std::string::npos) {
            name.erase(found, 1);
            found = name.find(" ");
        }
        filename.clear();
        filename = "md .\\" + std::to_string(Tones) + "Tones\\KeyMods\\{"
            + std::to_string(u + 1) + "}\\" + name + "\\";
        system(filename.c_str());

        Scale SV = ScalesVector[u][v];
        SV.Modify(0);

        for (auto NoteA : SV.GetiScale()) {



            ModeA = SV;
            unsigned ModeDeg = ModeA.GetiDeg(NoteA);
            ModeA.Modify(ModeDeg);

            mode.clear();
            // Here, it's okay to call GetMode with ModeDeg, since its iMode is 0.
            // That is, we have "SV.Modify(0);" SV is in root mode.
            mode = SV.GetMode(ModeDeg);

            found = mode.find(" ");
            while (found != std::string::npos) {
                mode.erase(found, 1);
                found = mode.find(" ");
            }

            std::string myfilename = ".\\" + std::to_string(Tones) + "Tones\\KeyMods\\{"
```

```
    + std::to_string(u + 1) + "}\\" + name + "\\" + mode + ".txt";

file.clear();
file.open(myfilename, std::fstream::out);

if (file.good()) {

    // 1. Compute Change of Density (decreasing)
    for (auto& NoteC : ModeA.GetiScale()) {
        Scale To_ScaleA = ModeA;

        unsigned norm = To_ScaleA.DeleteNote(NoteC);

        unsigned degs = To_ScaleA.GetDegs();
        std::pair<int, int> w{};
        unsigned m = 0;


        if (degs > 0) {



            w = knownu(ScalesVector[degs - 1], To_ScaleA.GetDef());
            Scale To_ScaleB = ScalesVector[degs - 1][w.first];
            auto ScA = To_ScaleA.Mode(0);
            for (unsigned x = 0; x < degs; x++) {
                To_ScaleB.Modify(x);
                std::set<unsigned> DiffA =
                    To_ScaleB.SetDifference(To_ScaleA.GetiScale());
                if (DiffA.empty()) {
                    m = x;
                    break;
                }
            }

        }
        else {
            w.first = -1;
            w.second = 0;
            m = -1;
        }

        unsigned monr = ModeA.GetiDeg(NoteC);

        file << "([" << std::to_string(v + 1) << "]"
            << ModeDeg + 1 << ") -> "
            << "[+" << ((Tones - (w.second)) % Tones)
            << "/-" << w.second << "]";
        file << " " << std::to_string(monr + 1) << "(X) {"
            << std::to_string(degs) << "} (["
            << std::to_string(w.first + 1) << "]"
            << std::to_string(m + 1) << ")" << std::endl;

    }




    // 2. Compute Changes of Key, Scale and Mode
    for (unsigned Key : Gamut) {

        for (Scale ScaleB : ScalesVector[u]) {
            Scale MyScaleB = ScaleB;
            MyScaleB.Modify(0);
            MyScaleB.ChangeKey(Key);
            for (auto NoteB : MyScaleB.GetiScale()) {
                ModeB = MyScaleB;

```

```
            ModeB.Modify(ModeB.GetiDeg(NoteB));


            std::set<unsigned> DiffA =
                ModeA.SetDifference(ModeB.GetiScale());
            std::set<unsigned> DiffB =
                ModeB.SetDifference(ModeA.GetiScale());

            if (DiffA.size() == 1 && DiffB.size() == 1) {

                Resolution = (int)(*DiffB.begin()) -
                    (int)(*DiffA.begin());

                file << SV.GetMode(ModeDeg) << " -> "
                    << "[+" << int(Key) << "/-"
                    << ((unsigned int(Tones) -
                        unsigned int(Key)) % Tones) << "]";
                file << " " << ModeA.GetiDeg((*DiffA.begin())) + 1;
                file << "(" << Resolution << ") {"
                    << std::to_string(SV.GetDegs()) << "} "
                    << MyScaleB.GetModeOf((Tones + NoteB - Key) % Tones);
                file << std::endl;

            }
        }
    }
}

// 3. Compute Change in Density (Increasing)

auto diff = Chromatic.SetDifference(ModeA.GetiScale());

for (auto& NoteB : diff) {
    std::pair<int, int> w{};

    Scale To_ScaleA = ModeA;

    To_ScaleA.AddNote(NoteB);
    auto degB = To_ScaleA.GetDeg(ModeA.GetNote(ModeDeg));
    To_ScaleA.Modify(degB);

    unsigned degs = To_ScaleA.GetDegs();
    w = knownu(ScalesVector[degs - 1], To_ScaleA.GetDef());

    Scale To_ScaleB2 = ScalesVector[degs - 1][w.first];
    To_ScaleB2.Modify(0);

    unsigned m = 0;
    if (degs == Tones) {
        m = 0;
    }
    else {
        for (unsigned x = 0; x < degs; x++) {
            To_ScaleB2.Modify(x);
            std::set<unsigned> DiffA =
                To_ScaleA.SetDifference(To_ScaleB2.GetiScale());

            if (DiffA.empty()) {
                m = x;
                break;
            }
        }
    }
    file << "([" << std::to_string(v + 1)
        << "]" << ModeDeg + 1 << ") -> "
        << "[+" << ((Tones - (w.second)) % Tones)
        << "/-" << w.second << "]";
    file << " A(" << std::to_string(NoteB) << ") {"
```

```
                            << std::to_string(degs) << "} ([" 
                            << std::to_string(w.first + 1) << "]"
                            << std::to_string(m + 1) << ")" << std::endl;

                    }

                    file.close();


                }
                else std::cout << "not good" << std::endl;
            }
        }
    }

    return 0;
}

// Function to save ModAtlas to a file
bool SaveModAtlas(const std::string& filename) {
    std::ofstream ofs(filename, std::ios::binary);
    if (!ofs) {
        std::cerr << "Failed to open file for saving ModAtlas." << std::endl;
        return false;
    }
    // Write the number of entries in ModAtlas
    size_t mapSize = ModAtlas.size();
    ofs.write(reinterpret_cast<const char*>(&mapSize), sizeof(mapSize));
    // Iterate over ModAtlas
    for (const auto& [key, vec] : ModAtlas) {
        // Write AtlasKey: three unsigned ints
        unsigned key0 = std::get<0>(key);
        unsigned key1 = std::get<1>(key);
        unsigned key2 = std::get<2>(key);
        ofs.write(reinterpret_cast<const char*>(&key0), sizeof(key0));
        ofs.write(reinterpret_cast<const char*>(&key1), sizeof(key1));
        ofs.write(reinterpret_cast<const char*>(&key2), sizeof(key2));
        // Write vector size
        size_t vecSize = vec.size();
        ofs.write(reinterpret_cast<const char*>(&vecSize), sizeof(vecSize));
        // For each AtlasVal in the vector
        for (const auto& val : vec) {
            // Serialize first element: std::string
            const std::string& strVal = val.first;
            size_t strLen = strVal.size();
            ofs.write(reinterpret_cast<const char*>(&strLen), sizeof(strLen));
            ofs.write(strVal.data(), strLen);
            // Serialize scalet: pair<string, unsigned>
            const std::string& op = val.second.first;
            size_t opLen = op.size();
            ofs.write(reinterpret_cast<const char*>(&opLen), sizeof(opLen));
            ofs.write(op.data(), opLen);
            unsigned transposition = val.second.second;
            ofs.write(reinterpret_cast<const char*>(&transposition), sizeof(transposition));
        }
    }
    return true;
}

// Function to load ModAtlas from a file
bool LoadModAtlas(const std::string& filename) {
    std::ifstream ifs(filename, std::ios::binary);
    if (!ifs) {
        std::cerr << "Failed to open file for loading ModAtlas." << std::endl;
        return false;
    }
    ModAtlas.clear();
    // Read the number of entries in the map
```

```
        size_t mapSize = 0;
        ifs.read(reinterpret_cast<char*>(&mapSize), sizeof(mapSize));
        for (size_t i = 0; i < mapSize; i++) {
            // Read AtlasKey: three unsigned ints
            unsigned key0, key1, key2;
            ifs.read(reinterpret_cast<char*>(&key0), sizeof(key0));
            ifs.read(reinterpret_cast<char*>(&key1), sizeof(key1));
            ifs.read(reinterpret_cast<char*>(&key2), sizeof(key2));
            AtlasKey key = std::make_tuple(key0, key1, key2);
            // Read the vector size
            size_t vecSize = 0;
            ifs.read(reinterpret_cast<char*>(&vecSize), sizeof(vecSize));
            std::vector<AtlasVal> vec;
            for (size_t j = 0; j < vecSize; j++) {
                // Read string for first element of AtlasVal
                size_t strLen = 0;
                ifs.read(reinterpret_cast<char*>(&strLen), sizeof(strLen));
                std::string strVal(strLen, '\0');
                ifs.read(&strVal[0], strLen);
                // Read scalet: first element string and unsigned transposition
                size_t opLen = 0;
                ifs.read(reinterpret_cast<char*>(&opLen), sizeof(opLen));
                std::string op(opLen, '\0');
                ifs.read(&op[0], opLen);
                unsigned transposition = 0;
                ifs.read(reinterpret_cast<char*>(&transposition), sizeof(transposition));
                scalet s = std::make_pair(op, transposition);
                vec.push_back(std::make_pair(strVal, s));
            }
            ModAtlas[key] = vec;
        }
        return true;
    }


    // 5. Populate the Atlas.
    int PopulateAtlas() {
        //*********************************
        // Here, we populate our atlas
        // This is, technically our second routine. But it depends on the data
        // computed by the ModMaps routines.
        // So, we have a hierarchy:
        //
        // 1. Construct the scales by their fundamental intervals.
        // 2. Generate the scale list.
        // 3. Create the Scale objects
        // 4. Compute the Modulation Maps.
        // 5. Populate the Atlas.
        //
        // Each step depends on the previous.
        // Once we have the atlas, the engine is almost ready and has started.
        // But there is another step we must take, but that's in SoundLab.
        // It is to generate the 12-TET Spectrum, and some goodies, like
        // interval circles.
        // We also need a grammar of rhythm.
        // We want real-time audio recognition for harmonization.
        // In the Language section, we want to define and describe patterns as function of an event?
        // We definitely want to describe a groove.

        std::string buf;

        ModAtlas.clear();
        for (unsigned long long u = 0; u < ScalesVector.size(); u++) {

            for (unsigned i = 0; i < ScalesVector[u].size(); i++) {

                for (unsigned j = 0; j < ScalesVector[u][i].GetDegs(); j++) {

```

```cpp
                // [(Notes,From,Operation)] = [(Next,Transposition)];
                std::filesystem::path ScaleDegMod = ".\\" + std::to_string(Tones)
                    + "Tones\\ScaleDegMods\\";
                std::string ScaleDegFrom;

                // This filename needs to exist before opening.
                std::string myfilename = ".\\" + std::to_string(Tones)
                    + "Tones\\KeyMods\\{" + std::to_string(u + 1)
                    + "}\\[" + std::to_string(i + 1) + "]\\([" + std::to_string(i + 1)
                    + "]" + std::to_string(j + 1) + ")" + ".txt";
                std::string mymode;
                std::string mykey;
                std::string myops;
                std::string nextdeg;
                std::string nextmode;
                unsigned tokey;
                std::ifstream fin;
                fin.open(myfilename);
                if (!fin.is_open()) std::cout << "Failed to open " << myfilename << std::endl;
                else { //The Chromatic Scale has no modulations!

                    while (fin.good() && !std::filesystem::is_empty(myfilename)) {
                        buf.clear();
                        // Here, when we populate the Atlas, we have to read the file correctly.
                        std::getline(fin, buf, '>');

                        buf.clear();
                        std::getline(fin, buf, '[');
                        std::getline(fin, mykey, ' ');

                        buf.clear();
                        unsigned c = 0;
                        while (mykey[c] != '/') {
                            if (mykey[c] == '0') buf = buf + '0';
                            else if (isdigit(mykey[c])) buf = buf + mykey[c];
                            c++;
                        }
                        if (buf == "") tokey = 0;
                        else tokey = stoi(buf);


                        fin >> myops;
                        std::getline(fin, buf, ' ');
                        std::getline(fin, nextmode);

                        auto mytup = std::make_tuple(unsigned(u + 1), i + 1, j + 1);

                        scalet AtVal = std::make_pair(myops, tokey);
                        auto myres = std::make_pair(nextmode, AtVal);
                        ModAtlas[mytup].push_back(myres);

                        buf.clear();

                    }
                }
                fin.close();
            }



        }
    }
    std::cout << "ModAtlas Size: " << ModAtlas.size() << std::endl;
    SaveModAtlas(".\\ModAtlas.mem");
    return 0;

}
```

```cpp

    void Start2() {
        Gamut.clear();
        for (unsigned i = 0; i < Tones; i++) {
            Gamut.insert(i);
        }

        ConstructScales();
        // This only needs to be run once. And when computing modulations.
        GenerateScaleList();
        CreateScales();
        //These three only need to be run once;
        ComputeKeyModulations();

        if (LoadModAtlas(".\\ModAtlas.mem")) {
            std::cout << "ModAtlas " << ModAtlas.size() << std::endl;
        }
        else {
            PopulateAtlas(); // Size of 25344 for 12 Tones
        }
    }


    /***********************************************************
*   Here, we have certain procedures for our scale objects.
*/

    int GetNextFrom() {
        /****************************************************************
        *   Here, we have an interface to get the possible modulations from a certain
        *   scale-mode.
        *   Works!
        */
        unsigned FromTones;
        unsigned FromScale;
        unsigned FromMode;
        unsigned FromKey;
        unsigned boof = 0;
        bool NEXT = false;
        AtlasKey From{};
        scalet ScaleMode;
        std::cout << "Get Next ? ";
        std::cin >> NEXT;
        while (NEXT) {
            std::cout << "Enter Tones Scale Mode Key: ";
            std::cin >> FromTones >> FromScale >> FromMode >> FromKey;
            From = std::make_tuple(FromTones, FromScale, FromMode);

            std::vector <AtlasVal> NeXT{};

            NeXT = ModAtlas[From];

            for (auto n : NeXT) {
                std::cout << n.first << " " << n.second.first << " "
                    << (FromKey + n.second.second) % Tones << std::endl;
            }
            std::cout << std::endl;

            std::cout << "Get Next ? ";
            std::cin >> NEXT;
            ScaleMode.first.clear();
        }
        return 0;
    }

    int GetPreviousTo() {
        /****************************************************************
        *   Here, we have an interface to get the possible modulations to a certain
```

```
        *    scale-mode.
        *    Works!
        */

        unsigned ToTones = 0;
        unsigned ToScale = 0;
        unsigned ToMode = 0;
        unsigned ToKey = 0;

        std::vector <std::pair <AtlasKey, scalet>> PREV;
        bool previous = false;
        std::cout << "Get Previous? ";
        std::cin >> previous;
        while (previous) {

            std::cout << "Enter Tones Scale Mode Key: ";
            std::cin >> ToTones >> ToScale >> ToMode >> ToKey;

            AtlasKey MyVal = std::make_tuple(ToTones, ToScale, ToMode);

            PREV = prev(ModAtlas, MyVal, Tones);

            for (auto& P : PREV) {
                // {Notes} ([Scale]Mode): NoteOperation, Transposition
                std::cout << EncodeKey(P.first) << ": " << P.second.first << ", "
                    << (Tones + ToKey - P.second.second) % Tones << std::endl;

            }
            std::cout << std::endl;
            std::cout << "Get Previous? ";
            std::cin >> previous;
        }

        return 0;
    }


    Scale KeyToScale(AtlasKey MyKey) {
        Scale MyScale;
        MyScale = ScalesVector[get<0>(MyKey) - 1][get<1>(MyKey) - 1];
        MyScale.Modify(get<2>(MyKey) - 1);
        return MyScale;
    }
    Scale KeyToScale2(AtlasKey MyKey) {
        Scale MyScale;
        MyScale = ScalesVector[get<0>(MyKey)][get<1>(MyKey)];
        MyScale.Modify(get<2>(MyKey));
        return MyScale;
    }
    void GetVectorDistance() {
        unsigned FromTones;
        unsigned FromScale;
        unsigned FromMode;
        unsigned FromKey;
        scalet From;
        scalet To;
        std::cout << "EnterFrom Notes Scale Mode Key: ";
        std::cin >> FromTones >> FromScale >> FromMode >> FromKey;
        From.first = encode(FromTones, FromScale, FromMode);

        From.second = FromKey;
        Scale FromS = KeyToScale(decode(From.first));
        FromS.ChangeKey(From.second);
        //FromS.Modify(FromMode);



        std::cout << "EnterTo Notes Scale Mode Key: ";
```

```
        std::cin >> FromTones >> FromScale >> FromMode >> FromKey;
        To.first = encode(FromTones, FromScale, FromMode);
        To.second = FromKey;
        Scale ToS = KeyToScale(decode(To.first));
        ToS.ChangeKey(To.second);

        std::vector<unsigned> curr;
        std::vector<unsigned> next;

        auto FiS = FromS.GetiScale();
        for (auto Note : Gamut) {
            auto MyNote = FiS.find((FromS.GetKey() + Note) % Tones);
            if (MyNote != FiS.end()) {
                curr.push_back(*MyNote);
                std::cout << *MyNote << ", ";
            }
        }
        std::cout << std::endl;
        auto TiS = ToS.GetiScale();
        for (auto Note : Gamut) {
            auto MyNote = TiS.find((ToS.GetKey() + Note) % Tones);
            if (MyNote != TiS.end()) {
                next.push_back(*MyNote);
                std::cout << *MyNote << ", ";
            }
        }
        std::cout << std::endl;
        std::cout << "Distance: " << VecDist(curr, next) << std::endl;

    }
    struct CompareDist {
        bool operator()
            (const std::tuple<scalet, double, scalet>& lhs,
                const std::tuple<scalet, double, scalet>& rhs) const {
            return std::get<1>(lhs) < std::get<1>(rhs);
        }
    };
    bool IsAllInfinity(std::vector<std::tuple<scalet, double, scalet>> Visited) {
        for (auto& i : Visited) {
            if (get<1>(i) != std::numeric_limits<double>::infinity()) {
                return false;
            }
        }
        return true;
    }
    std::vector<scalet> DIJKSTRA(graph Atlas, scalet root, scalet goal, unsigned Tones) {

        double CurrentDistance = 0;
        double PathDist = std::numeric_limits<double>::infinity();
        Scale Root;
        Scale Goal;
        Scale CurState;
        Scale NexState;

        scalet CurrentState;

        std::vector<std::tuple<scalet, double, scalet>> GraphTable;
        std::vector<std::tuple<scalet, double, scalet>> Visited;

        std::vector<std::pair<AtlasKey, unsigned>> NextStates;

        for (auto& Rule : Atlas) {

            for (unsigned Note : Gamut) {
                double distance;
                if (Rule.first == decode(root.first) && Note == root.second)
                    distance = 0;
                else distance = std::numeric_limits<double>::infinity();
```

```
                    GraphTable.push_back(std::make_tuple(
                        std::make_pair(EncodeKey(Rule.first), Note),
                        distance, std::make_pair("{0} ([0]0)", unsigned(0))));
                }

            }
            std::cout << "Unvisited: " << GraphTable.size() << std::endl;


            while (!GraphTable.empty()) {
                double leastdistance = std::numeric_limits<double>::infinity();
                CurrentDistance = leastdistance;
                for (auto& item : GraphTable) {
                    if (get<1>(item) < leastdistance) {
                        CurrentState = get<0>(item);
                        CurrentDistance = get<1>(item);
                    }
                }
                if (CurrentDistance == leastdistance) {
                    break;
                }
                if (CurrentState != goal) {
                    std::cout << "current state: " << CurrentState.first << " "
                        << CurrentState.second << std::endl;
                    NextStates.clear();


                    NextStates =
                        next(
                            ModAtlas,
                            decode(CurrentState.first),
                            CurrentState.second,
                            Tones);

                    CurState = KeyToScale(decode(CurrentState.first));
                    CurState.ChangeKey(CurrentState.second);
                    std::vector<unsigned> curr;
                    auto FiS = CurState.GetiScale();
                    for (auto Note : Gamut) {
                        auto MyNote = FiS.find((CurState.GetKey() + Note) % Tones);
                        if (MyNote != FiS.end()) {
                            curr.push_back(*MyNote);
                        }
                    }

                    for (auto State : NextStates) {
                        NexState = KeyToScale(State.first);
                        NexState.ChangeKey(State.second);

                        std::vector<unsigned> next;


                        auto TiS = NexState.GetiScale();
                        for (auto Note : Gamut) {
                            auto MyNote = TiS.find((NexState.GetKey() + Note) % Tones);
                            if (MyNote != TiS.end()) {
                                next.push_back(*MyNote);
                            }
                        }


                        double PairDist = CurrentDistance + VecDist(curr, next);

                        for (auto& item : GraphTable) {

                            if (decode(get<0>(item).first) == State.first
                                && get<0>(item).second == State.second
```

```
                            && PairDist < get<1>(item)) {

                            auto tup = std::make_tuple(get<0>(item), PairDist, CurrentState);

                            item.swap(tup);
                            break;
                        }
                    }
                }

                for (auto it = GraphTable.begin(); it != GraphTable.end(); ++it) {
                    if (get<0>(*it) == CurrentState) {
                        Visited.push_back(*it);
                        GraphTable.erase(it);
                        std::cout << "Visited: " << Visited.size()
                            << ", GraphTable: " << GraphTable.size() << std::endl;
                        break;
                    }
                }

            }
            else {
                std::cout << "Terminate search. \n";

                for (auto it = GraphTable.begin(); it != GraphTable.end(); ++it) {
                    if (get<0>(*it) == CurrentState) {
                        Visited.push_back(*it);
                        GraphTable.erase(it);
                        std::cout << "Visited: " << Visited.size()
                            << ", GraphTable: " << GraphTable.size() << std::endl;

                        break;
                    }
                }
                break;
            }
        }
        // step 6
        std::cout << "Step 6.\n";
        std::vector<scalet> Progression;
        std::cout << "Goal: " << goal.first << " " << goal.second << std::endl;
        CurrentState = goal;
        std::cout << "Root: " << root.first << " " << root.second << std::endl;

        do {

            std::cout << "Current State: " << CurrentState.first
                << " " << CurrentState.second << std::endl;
            for (unsigned uns = 0; uns < Visited.size(); uns++) {
                if (get<0>(Visited[uns]) == CurrentState) {
                    Progression.push_back(CurrentState);
                    CurrentState = get<2>(Visited[uns]);
                    std::cout << get<0>(Visited[uns]).first << " "
                        << get<0>(Visited[uns]).second << std::endl;
                    auto it = Visited.begin();
                    Visited.erase(it + uns);
                    std::cout << "Progression: " << Progression.size() << std::endl;
                    break;
                }
            }

        } while (CurrentState != root);
        return Progression;
    }

    int SearchDijkstra() {
        /**********************************************
        *   This procedure searches for the shortest path in the Atlas
```

```
        *   between two pitched scale-modes by means of Dijkstra's
        *   Algorithm. Other algotithms may be used. Must expand into that.
        */

        unsigned FromTones;
        unsigned FromScale;
        unsigned FromMode;
        unsigned FromKey;
        bool search = false;
        unsigned boof = 0;
        std::cout << "Search Dijkstra\'s? ";
        std::cin >> search;
        while (search) {
            scalet From;
            scalet To;
            std::cout << "ModAtlas: " << ModAtlas.size() << std::endl;
            std::cout << "EnterFrom Tones Scale Mode: ";
            std::cin >> FromTones >> FromScale >> FromMode >> FromKey;

            From.first = encode(FromTones, FromScale, FromMode);
            From.second = FromKey;

            std::cout << "EnterTo Tones Scale Mode: ";
            std::cin >> FromTones >> FromScale >> FromMode >> FromKey;
            To.first = encode(FromTones, FromScale, FromMode);
            To.second = FromKey;

            std::vector <scalet> Trails = DIJKSTRA(ModAtlas, From, To, Tones);
            for (auto i : Trails) {
                std::cout << i.first << " " << i.second << std::endl;
            }
            std::cout << "Search Dijkstra\'s? ";
            std::cin >> search;
        }
        return 0;
    }
};
```