# Music within

# Conway's Game of Life

UNIVERSIDAD
POLITÉCNICA
DE MADRID


ETSI SISTEMAS
INFORMÁTICOS

**Final Degree Project**

Software Engineering and Technologies for Information Society

Luis Carlés Durá

DIRECTED BY:

ANA ISABEL LÍAS QUINTERO

# ACKNOWLEDGEMENTS

First of all, thank you very much, Ana. Neither of my two final degree projects would have gone ahead without you and that means a lot considering the circumstances. Thank you for not giving up and refining this naive's weird and wacky ideas to the end.

To my dear chubby Jaime, for all those afternoons solving together the contingencies that were presenting themselves without prior notice. Much of this project also bears your indelible signature. Despite the mediocrity and tedium, just knowing you already made all these years of career worthwhile. I don't regret a thing.

To Jennie, Lisa, Jisoo and Rosé: there is no word or concept that I know of that can encompass my gratitude and admiration for you. It is a true privilege to be a contemporary of yours and to be able to enjoy your talent. Anyone who knows how to read between the lines will be able to see that this work, being impregnated with my unconditional love for BLACKPINK, is a veiled ode to you.

너의 지역에 있는 블랙핑크

To my parents, I am unable to write these lines without silently sobbing. I know you'll be there for every success, failure, and action I take, and it feels like clever cheating with no risk incurred. I owe you absolutely everything that I am, and life will not reach me to thank you. For now, remind you that your son cannot love you more while he runs to give you a hug.

# ABSTRACT

It is hard to believe that, contrary to what logic suggests at first, there are games for zero players. If one tries to challenge the concept itself, the mere attempt to accept the existence of these games often leads to cognitive dissonance.

Conway's Game of Life is a cellular automaton that takes place on a board with infinite squares that, despite having very simple rules, is considered a universal Turing machine capable of modeling any computer program. Discovered by John Horton Conway in 1970, it was quickly accepted as a classic problem among the programming community, clearly influenced by the pop culture of the time.

Extremely limited resources are assumed. If you have a lot of people around, you would die from overpopulation and if you have practically no one, you would die from loneliness. If you are lucky and hit with unexpected kindness, some individuals may prevent your death. They seem like simple biological rules that come to shape life somewhere in the world, but in fact, those are the rules of the Conway’s Game of Life.

It seems that nature laughs at us, forcing us to consider very complex problems only to later discover that their solution was utterly simple and beautiful. But I think that in its eagerness to be unfathomable it ends up sinning with pride. And if mathematics represents its language, music represents his own voice, provoking emotions, stimulating creativity, and allowing us to respond to its blatant attempt to hide.

In this sense, this project constitutes a somewhat extravagant Python implementation of Conway's game of life and will study its relationship with music, creating a relationship between those graphic patterns in the game and a specific sound produced by a musical instrument. Even biasing the process, after stepping aside, the game will be able to compose music while performing its usual modus operandi. Once that first step is achieved, I will bend its will and possibilities to force it to model one of the themes that have resonated with me in the most recent times.

As the title track of BLACKPINK's new album, Shut Down represents a new horizon for the genre. Masterfully combines a famous violin concerto by Niccolò Paganini from the 19th century with the deep musical production related to modern trap. A curious fusion of such dissimilar and distant elements that has been a commercial success.

This work represents a challenge up to all these years studying software engineering, an opportunity to involve my vocation for mathematics and an homage to one of the pillars of my life, music.

**Keywords**: Conway, Python, Music.

# RESUMEN

Cuesta creer que, contrariamente a lo que sugiere la lógica en un principio, existan juegos para cero jugadores. Si se intenta desafiar el concepto en sí mismo, el mero intento de aceptar la existencia de estos juegos a menudo conduce a una disonancia cognitiva.

El Juego de la Vida de Conway es un autómata celular que se desarrolla en un tablero con infinitas casillas y que, a pesar de tener reglas muy simples, se considera una máquina de Turing universal capaz de modelar cualquier programa informático. Descubierto por John Horton Conway en 1970, rápidamente fue aceptado como un problema clásico entre la comunidad de programadores, claramente influenciados por la cultura pop de la época.

Se asumen recursos extremadamente limitados. Si tienes mucha gente alrededor, morirías por sobrepoblación y si no tienes prácticamente a nadie, morirías por soledad. Si tienes suerte y recibes una amabilidad inesperada, algunas personas pueden evitar tu muerte. Parecen simples reglas biológicas que podrían llegan a modelar la vida en algún lugar del mundo, sin embargo, esas son las reglas del Juego de la Vida de Conway.

Parece que la naturaleza se ríe de nosotros, obligándonos a considerar problemas muy complejos para luego descubrir que su solución es increíblemente simple y hermosa. Pero creo que en su afán de ser insondable acaba pecando de soberbia. Si las matemáticas representan su lenguaje, la música representa su propia voz, provocando emociones, estimulando la creatividad y permitiéndonos responder a su descarado intento de ocultarse.

En este sentido, este proyecto constituye una implementación en Python algo extravagante del juego de la vida de Conway y estudiará su relación con la música, creando una relación entre esos patrones gráficos en el juego y un sonido específico producido por un instrumento musical. Incluso sesgando el proceso, después de hacerme a un lado, el juego podrá componer música mientras realiza su habitual modus operandi. Una vez logrado ese primer paso, doblegaré su voluntad y posibilidades para forzarlo a modelar uno de los temas que me han resonado en los tiempos más recientes.

Como tema principal del nuevo álbum de BLACKPINK, Shut Down representa un nuevo horizonte para el género. Combina magistralmente un famoso concierto para violín de Niccolò Paganini del siglo XIX con la profunda producción musical del trap moderno. Una curiosa fusión de elementos tan disímiles y lejanos que ha sido un éxito comercial.

Este trabajo representa un reto a la altura de todos estos años estudiando ingeniería de software, una oportunidad para involucrar mi vocación por las matemáticas y un homenaje a uno de los pilares de mi vida, la música.

**Palabras clave**: Conway, Python, Música.

# INDEX


**1. INTRODUCTION ..... 5**

1.1. Project Goals ..... 6

1.2. Document Structure ..... 6

**2. CELLULAR AUTOMATA ..... 7**

2.1. Conway's Game of Life ..... 9

2.2. Patterns ..... 11

2.2.1. Still lifes (S.L) ..... 11

2.2.2. Oscillators (O) ..... 13

2.2.3. Spaceships (S) ..... 14

2.2.4. Methuselah (M) and other complex configurations ..... 19

2.3. Running some initial basic configurations ..... 23

**3. CONWAY'S GAME OF LIFE IMPLEMENTATION ..... 29**

3.1. Overview and Design decisions ..... 29

3.2. yawnoc.py ..... 35

3.3. actualize.py ..... 44

**4. SOUND AND MUSIC WITHIN CONWAY'S GAME OF LIFE ..... 49**

4.1. maths.py: association function between grid and sound ..... 49

4.2. music.py: from grid to sound ..... 60

4.3. Second Approach: from sound to grid ..... 66

4.3.1. BLACKPINK - 'Shut Down' ..... 66

4.3.2. Separating mp3 song in tracks with Demucs ..... 78

4.3.3. finder.py , keeper.py ..... 82

4.3.4. shutdown.py and results of the model ..... 90

**5. PLANNING AND METHODOLOGY ..... 99**

**6. SOCIAL AND ENVIRONMENTAL IMPACT: ETHICAL AND SOCIAL RESPONSIBILITY 101**

**7. CONCLUSIONS AND DEVELOPMENT PROPOSALS ..... 103**

**8. BIBILIOGRAPHY ..... 105**

# 1. INTRODUCTION

I remember recently walking in the Park Güell in Barcelona, under a surprisingly clear sky and being overdressed for that temperature. I found no peace of mind, suffering the aftermath of last night's concert and having certainly lived through one of the most cathartic experiences of my life. My dramatic arc at that time, led by an antisocial, depressive and nihilistic trend, was finally coming to an end. All my doubts vanished in that instant, evolving into fierce decisions and allowing me to accept the call to adventure.

The only negative loophole of that turning point was that everything seemed trivial to me from that point forward. The standard and my expectations had skyrocketed, and despite being surrounded by so much art and beauty, I was not able to truly enjoy it. Partially abandoning logic and reasoning, I thought that I should relive and make the most of the experience I lived, forgetting that those memories were already part of me. The ecstasy left me dizzy at times, but it was precisely then when something unexpected happened.

I was dodging some passers-by avoiding slipping when I tuned my ear and recognized a melody that was very familiar to me. Beneath one of the park's most recognizable aqueducts, which seems to be on the verge of collapsing at any moment, a guitarist was playing Leonard Cohen's Hallelujah surrounded by some onlookers who were drawn to his virtuosity like me. Despite being in a hurry to catch the train back, I could not do more than sit and admire what was in front of me. A boy crawled on the ground to slightly separate the guitarist's backpack, which was already wet from the man's tears. Too engrossed in his performance, he seemed entranced as he improvised once the song was over. The music gave me everything last night, took my peace briefly that morning, and gave it back to me at noon, giving me a lesson I may never forget.

Being able to appreciate such an intimate and emotional moment, full of emptiness and calm, was the cure for the state of ecstasy in which I found myself, at that moment I only began to feel emotion for what was to come, not despair to replicate what had already happened. Everything seemed to have tonality while I was trapped in extreme synesthesia, wondering what melodies the mosaics in the square suggested or what chords could be extracted from the colors of the birds that stole bread crumbs from the benches.

Despite being a gesture that was not at all up to his performance, I was left without any coin after giving that unknown guitarist a wink, who returned it to me with confidence despite never having seen me. Already sitting on the train back, while slowly returning to the harsh reality, I resumed the brainstorming about my second project and the cellular automata that I had left half a day before. As a final piece of pure inspiration, a question crossed my mind: "How does Conway's game of life sound?"

## 1.1. **Project Goals**

The main objective of the project is to develop a software product that meets my expectations and that implements all the required functionality in a clear, concise, and easily understandable way. So that, even in a more superficial way, a user outside the developer profile can understand some parts of the code.

In addition, and as a specific goal, explain the intrinsic importance of the theory of cellular automata and its possibilities of application to other areas, particularly breaking down the barrier between music, mathematics and computing. Exploring its possible behavior as a visualizer, a graphic way to represent music in support of the hearing-impaired community.

Another of my goals is to get used to planning software development using the strategies learned over the years, with the intention of avoiding delays, achieving a quality product, and applying correct risk and contingency management.

Defy the simplicity paradox by establishing some very simple rules and features that hold great potential against all odds, finishing the project taking to the limit the possibilities of the developed environment.

And, ultimately, to reconcile myself with software engineering. In one of my worst moments, it unassumingly posited itself as the only possible escape from frustration and failure. Even though I was never overly excited, becoming desperately bored in recent years, perhaps the real culprit is myself. I was not able to experience any catharsis with knowledge, in any of its branches, but if I now intend to selfishly take advantage of my learning to start a new stage away from the country that saw me grow up, I should at least respect what makes it possible.

## 1.2. **Document Structure**

The document follows the order of contents presented in the index, distinguishing 5 main sections, and most with specific subsections. Most of the definitions and concepts that are presented are not numerically related to their section, since they will not require traceability later.

However, the images or diagrams are subject to possible calls. That is why they respect a numbering system that corresponds to a heading that presents the specific section in which they are, preceded by a cardinal, if there are more images in the section. For aesthetic reasons, the screenshots related to the code of the Python files are exempt from respecting this numbering since they are explained as soon as they appear.

# 2. CELLULAR AUTOMATA

Mathematics represents an interconnected network of different areas. Particularly in automata theory, a cellular automaton is just a standard approach to model some scenarios using an a priori infinite grid of cells. Mathematicians and engineers have found several applications for these cellular spaces in various knowledge sectors, such as: physics, microbiology, or pattern recognition among others.

These idealizations assume discrete space and time for the system of interest, and also start under the assumption that physical quantities take on a finite set of discrete values. Understanding that each cell encompasses a discrete variable, a "state" of the cellular automata is completely defined by taking into account all the particular values of those variables. The initial state is given when $t = 0$, but a cellular automaton is capable of dynamically evolve in discrete time steps.

The variables are updated simultaneously by examining its neighborhood and applying a definite set of "local rules" that does not usually changes over time. In the late 40's or early 50's, the automaton concept was initially discovered by Stanislaw Ulam and John Von Neumann while they were trying to imitate the way biological cells auto-replicate themselves in a computerized machine. In my opinion, another example of the complexity paradox in nature can be found here. At first, imitating something as crucial as the reproduction and the behavior of living beings, can be wrongly prejudge as a deeply complex task. When, in fact, some of the most simple and beautiful mathematical solutions are precisely the simplest one anyone could ever came with.

At that time, the very first cellular automaton was a two-dimensional one, made up of regular squares, in which each cell can take one of 29 possible values for its variable. The neighborhood was defined by Neumann as the figure 2.1 shows bellow.

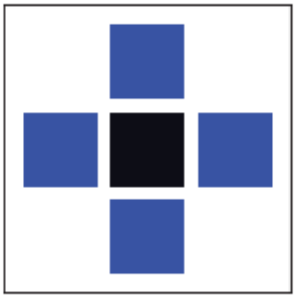


Figure 2.1: Neumann's Neighborhood

Unfortunately, the fact that technology was not up to it yet, and given also the complexity of that first model, there was no surging interest and research around cellular automata until some decades after, when Conway's Game of Life was presented.

From that point forward, given the success of the game and its ubiquity between the programming communities, some researchers started the trend of working with cellular automata in order to model some physical problems and trying to accomplish the smallest Turing Machine. In his book: *A New kind of Science*, Stephen Wolfram gave one precise classification for cellular automata according to not only their behavior but their tendency to create recognizable patterns. This classification can be summarized in these four cases:

- **Class 1:** The initial state of the automaton tends to global stability in all cells quickly, mimicking the usual entropic way that some physics systems have, where inertia prevails.

- **Class 2:** The automaton sooner or later tends to conform itself into a grid of stable structures and static patterns. Those structures are recursive and repeat themselves over time.

- **Class 3:** Unlike in the previous class, the structures and patterns generated by this type of cellular automata are now chaotic and random. The possible stable patterns that could initially form in the grid disappear, being destroyed literally by the uncontrolled nature of new ones.

- **Class 4:** From my point of view, the most interesting, infrequent and susceptible to being applied in engineering class of automaton. One cellular automata of this type can generate complex structures capable of survive and evolve for large periods of time, even sub-generating dynamic patterns that give us the idea they are actually travelling around the grid. Usually but not forcefully they could even terminate in a typical Class2 stable state.

In short, every automaton can be described informally as:

- A **grid** made up with cells as an idealization of an X-Dimensioned infinite space.

- A set of possible values/**states** for each of the intrinsic variables in the cells.

- A definite notion of adjacency or **neighborhood** management.

- A set of **local rules** in charge of determining the updated value according to the neighborhood.

And assuming **Principle of Homogeneity** about all the previous concepts.

## 2.1. **Conway's Game of Life**

In October 1970, the nowadays most famous cellular automaton was created by John Horton Conway and published in the *Mathematical games* magazine by Martin Gardner. Conway's Game of Life constitutes a two-dimensional square-tiling cellular automaton of Class4 in Stephen Wolfram classification.

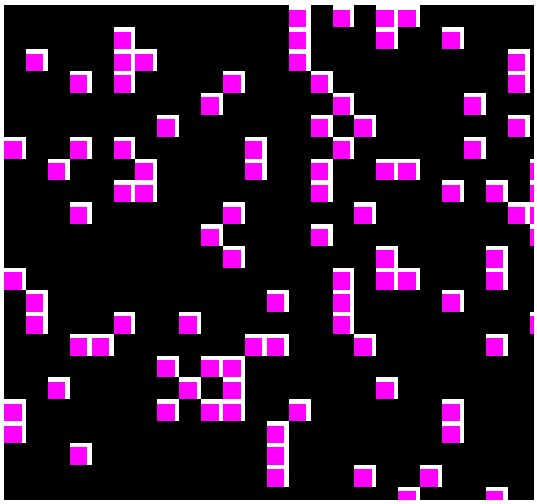


Figure 2.1.1: Traditional Representation of Conway's Game of Life

Unlike the previously commented automaton, the Game of Life works under the most binary possible assumption for each cell's variable value. One cell can either be alive or dead, leaving then two possible states. In order to finish with the theoretical definition of the automata it only remains to explain how the neighborhood is examined and which set of "local rules" is the system using in order to update the value of the cell in the next time step. On the one hand, the considered neighborhood is a range-1 Moore one, taking into account as well the diagonal cells at unit distance, as is shown in the figure 2.1.2.

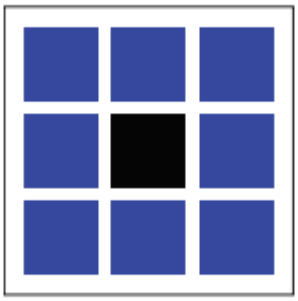


Figure 2.1.2: Range-1 Moore's Neighborhood

On the other hand, although a more rigorous definition using group theory would be far more interesting, for now the set of local rules that endures the Conway's Game of Life is the following:

- An alive cell (assigned value: 1) will remain with life if it has exactly 2 or 3 living cells in its neighborhood.

- An alive cell (1) will die (assigned value: 0) in the next time step due to overpopulation if it has strictly more than 3 living cells in its neighborhood.

- An alive cell (1) will die (0) in the next time step due to loneliness if it has strictly less than 2 living cells in its neighborhood.

- A dead cell (0) will revive (1) in the next time step if it has exactly 3 living cells in its neighborhood, as if by reproduction.

One might come to think that the game of life represents nothing more than a classic programming problem. I myself fell into that boring and surely not entertaining dead-end when I first came across this automaton years ago. Despite that first impression, my epiphany arrived at the same time that I discovered the mathematics behind it, one unexpected miscellaneous of Group and Game Theory with an outstanding potential. The mere idea of being able to force patterns that mimic the behavior of traditional logic gates is hopeful and robust enough to consider this game a universal Turing Machine. A simple and innocent example can be seen in the following figure, where the Droste effect is presented with the game of life.

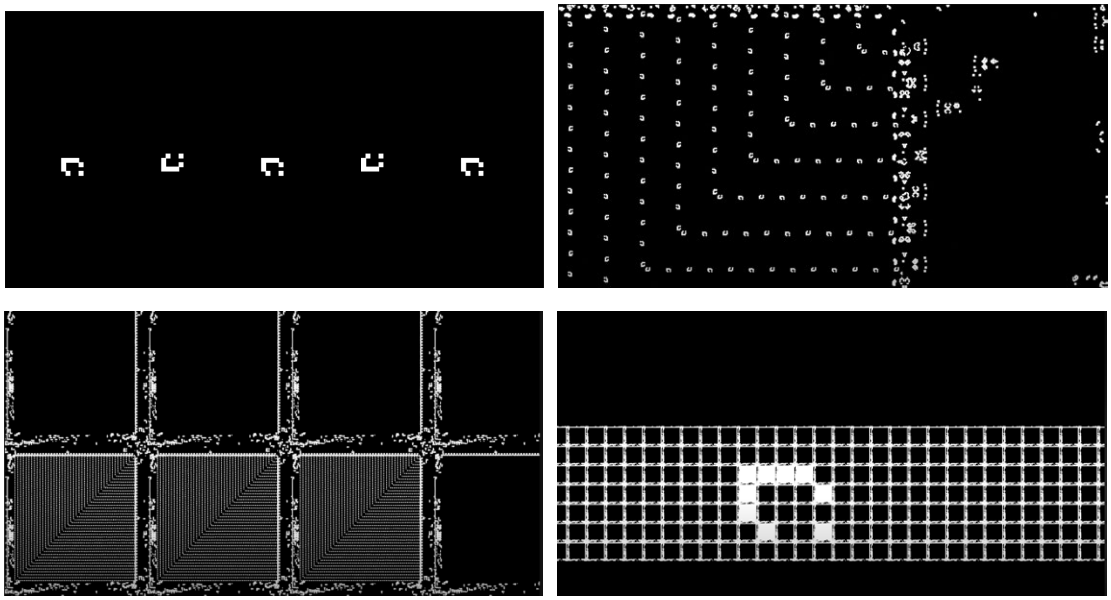


Figure 2.1.3: Droste effect in Conway's Game of Life

But in order to be able to appreciate the hidden power of this, the starting point can be found by classifying the different patterns to which the game entropically tends.

## 2.2. **Patterns**

### 2.2.1. ***Still lifes (S.L)***

The most stable patterns in game of life are called still lifes, and this is so because they do not evolve in the next time step and remains with the same exact shape. Also known as oscillators with period 1, still life patterns are usually assumed to be infinite and non-empty.

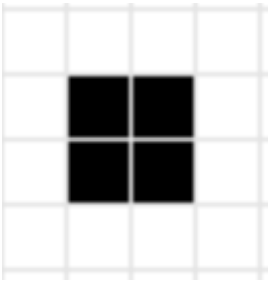


Figure 2.2.1.1: The “Block” the most common S.L

Depending on its number of separated “islands” (connected components), still lifes can be sorted into two main subgroups:

**Strict still lifes:** it is a still life that is either fully connect or it is a group of agglutinated islands forming a symbiotic system in which removing any one of them will affect the stability of the entire system. In the second case, any island by itself does not form a still life being stable.

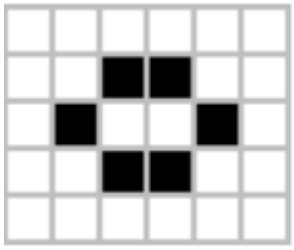


Figure 2.2.1.2: “Beehive” as Strict S.L

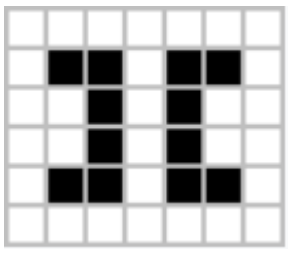


Figure 2.2.1.3: “Mirrored table” as Strict S.L

**Pseudo still lifes:** still lifes that consist of two or more islands that do not necessarily need each other in order to remain stable. This system in fact can be partitioned into non-interacting sub-patterns that are indeed another Pseudo S.L or, in the last atomic instance, a strict still life. But as prerequisite, there must be at least one dead cell that has more than three living cells around in the overall pattern but has less than three in the sub-patterns. This little condition automatically excludes any still lifes consisting in stable islands that are too far ahead between them.

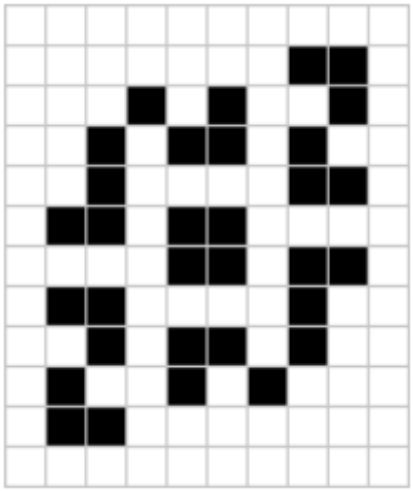


Figure 2.2.1.4: “Tripe pseudo still life”

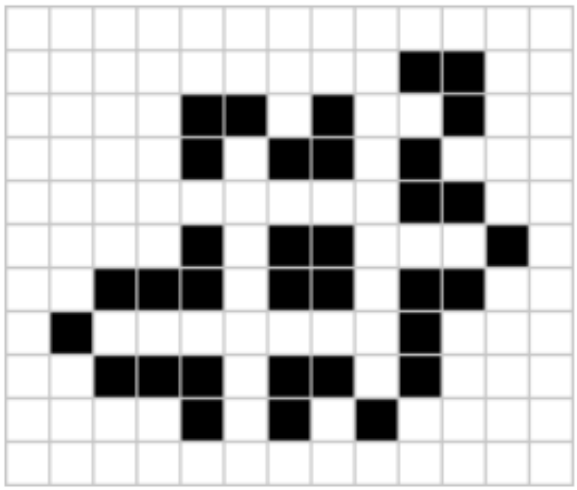


Figure 2.2.1.5: “Quad pseudo still life”

As an extra comment, note that a pseudo still life cannot be divided into more than four independent and stable sub-patterns due to the consistency of the Four-Color Theorem.

### 2.2.2. ***Oscillators (O)***

In Conway's Game of Life, some recursive patterns that are predecessors of themselves are called oscillators. In other words, after a definite number of generations, the initial state of the pattern is again formed in the same position on the grid. The most known oscillator is my forever-utterly-loved-by-far pattern for its name: the Blinker.

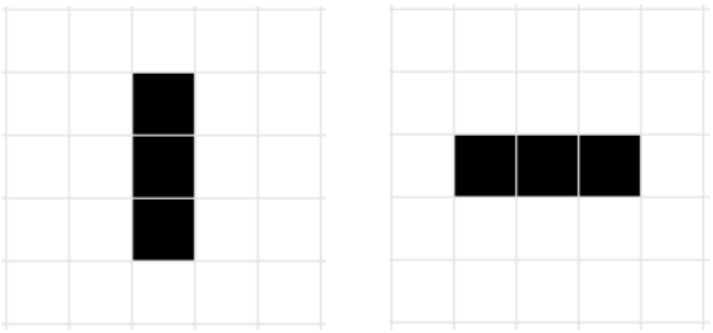


Figure 2.2.2.1: "Blinker" the most common O

Every oscillator has three fundamental parameters from which they can be classified, and these are:

- **Period:** automatically defined by the name, it is the number of iterations that must occur before the initial state of the oscillator is again reached.

- **Stator:** it is the precise set of cells which remain alive throughout the whole period.

- **Rotor:** it is the precise set of cells which really oscillates, changing its state in some of the generations along the period.

### 2.2.3. ***Spaceships (S)***

Every oscillator that changes its position after its period is called spaceship. These patterns can apparently give the sensation of movement in Conway's Game of Life due to this property. In fact, under the rules of group theory, what is really happening is that some cells are committing a translation through the grid, in other words, all coordinates of the points defining the entity are modified by adding the same vector quantity. The length of this vector is called the distance of the translation, considering that the distance between two neighbour cells is 1. The most known spaceship and the responsible of creating all the logic traditional gates is the "Glider".

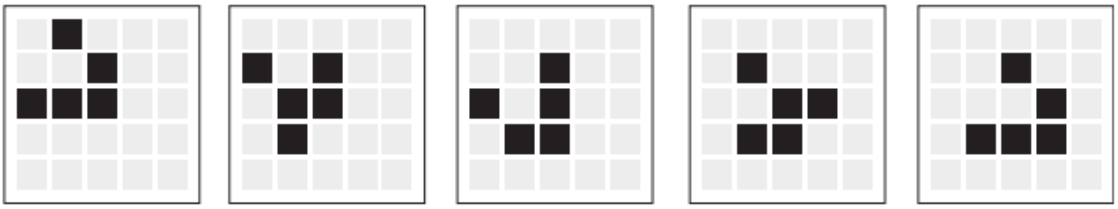


Figure 2.2.3.1: Execution of one "Glider", the most common S

The new parameter that appears on scene and must be highlighted is the associated **speed** for the spaceship. It consists of one numerical relationship between the distance of translation and the period of the spaceship expressed in terms of $c$, which comes to represent the metaphorical maximum possible "speed of light":

$$speed(S) = \frac{\#length\ of\ translation * c}{T}$$

The previous equation gives us for example the glider's speed which can be expressed as:

$$speed(Glider) = \frac{1 * c}{4}$$

what holds because the original location of the glider is translated one cell diagonally under a period of four generations.

There are some different classifications that can be applied to these patterns, but for the purposes of this project and particularly this "previous concepts" section, it will be enough just mention two of the most relevant ones.

The first type is applicable to all spaceships, not only for elementary ones but engineered too and it pays attention to the movement nature. On the other hand, the second one is more related to self-interaction between spaceships that built up one more colossal spaceship susceptible of replicate itself later on.

*Universal classifications*

**Flippers:** spaceship that forms its mirror image hallway through its period. These are also known as Glide-symmetric spaceships because they undergo reflection and translation simultaneously.

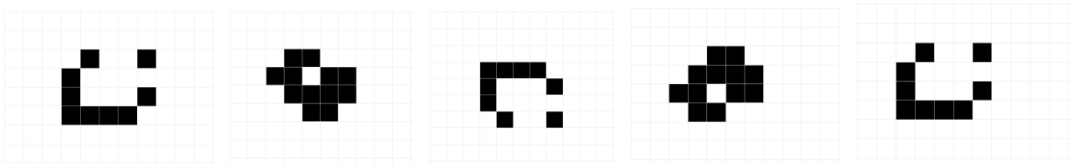


Figure 2.2.3.2: "Lightweight Spaceship", smallest orthogonal flipper

**Non-monotonic spaceship:** An S is said to be non-monotonic if it's "leading edge" (configuration on the front row of an active region) falls back in some generations, not necessarily moving forward or staying where it was.

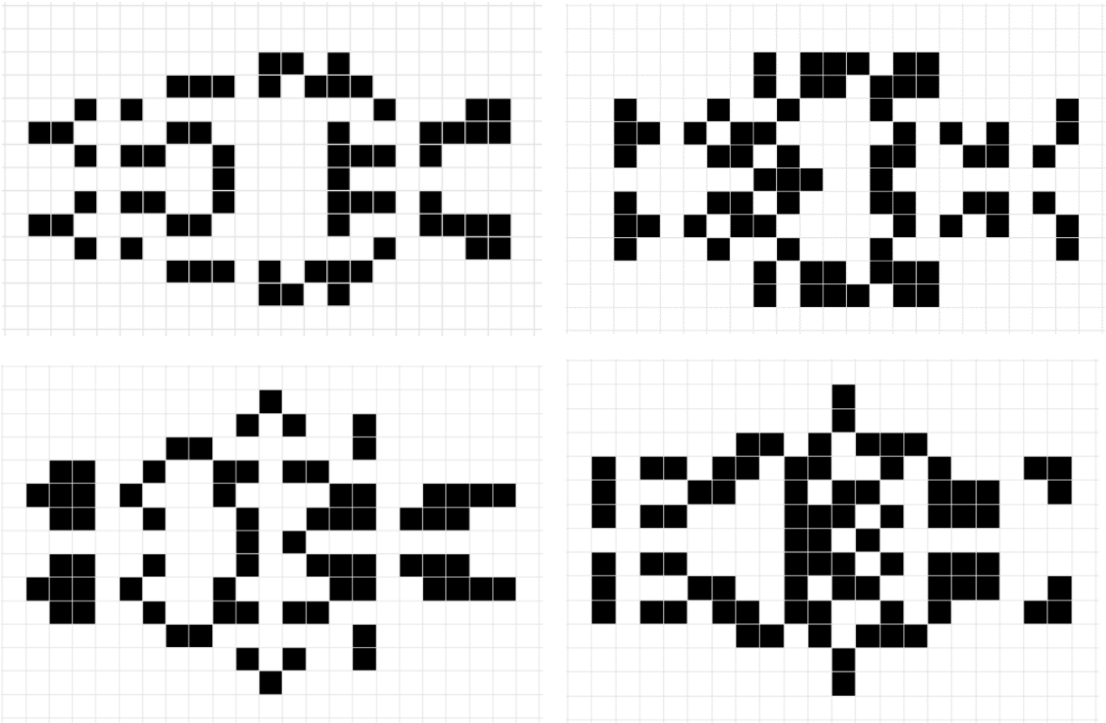


Figure 2.2.3.3: "Non-monotonic Spaceship 1", period 4 N-M S

**Oblique spaceship:** spaceship that does not move orthogonally or diagonally. The numerator in the previous equation for speed it is not an integer but a rational that expresses a combination of both movements. The most known subtype of an oblique spaceship is the Knightship, an oblique spaceship of type (2m, m), that is an S that moves two cells horizontally for each cell it moves vertically.

**Standard spaceships:** original spaceships that have been known since 1970, including: Glider, Lightweight spaceship (both previously shown), Middleweight spaceship or Heavyweight spaceships. A non-standard S is any other not commented here.

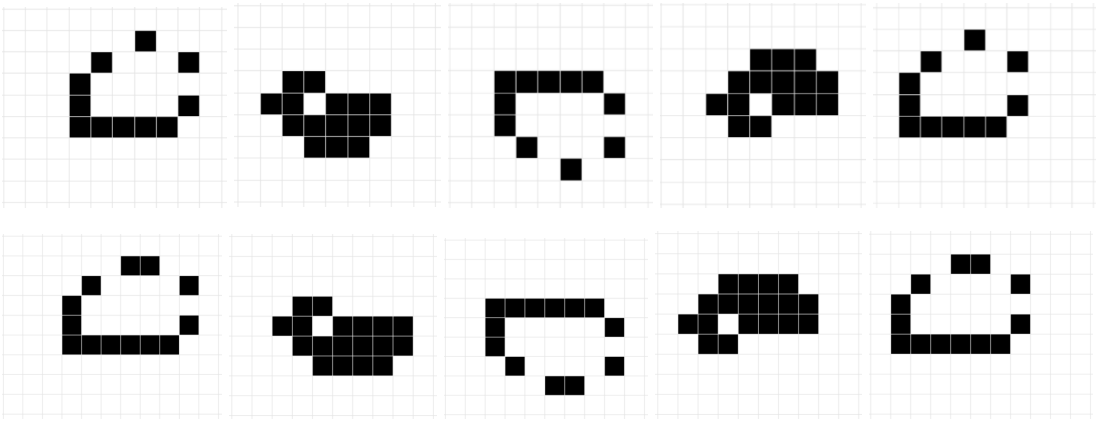


Figure 2.2.3.4: Up: “Middleweight S”, Down: “Heavyweight S”

**Edge-repair spaceships:** spaceship which edge, not necessarily its leading edge, leaves no “spark” (collection of cells periodically thrown away by an S) behind. Despite that feature, they still can repair certain types of damages to themselves, even rebuilding crucial structures for the generations to come.

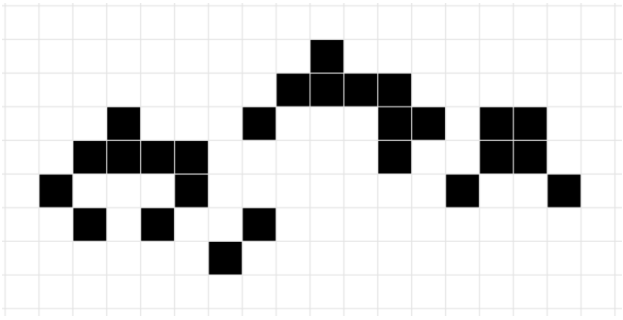


Figure 2.2.3.5: Initial state for “Edge-repair spaceship 1”

**Smoking ships:** spaceship which edge, not necessarily its leading edge, leaves smoke-type sparks behind. Managed correctly, if these smokes get past the edge and boundaries of the spaceship itself, they can be used to perturb other objects in the grid. Provoking controlled interactions between different sections, which are the basis of vanishing/duplicating gliders and therefore the first step in order to compute logic-gates-based programs.

**Flotilla:** a spaceship composed of several individually smaller spaceships. The set can move through the grid together as a collective, but even if some partitions need the group in order to survive, they do not really interact with each other. As it is shown in the figure down below, the "Flotilla 1" is composed by two Heavyweight spaceships escorting a long overweight S which is the center leading the group.

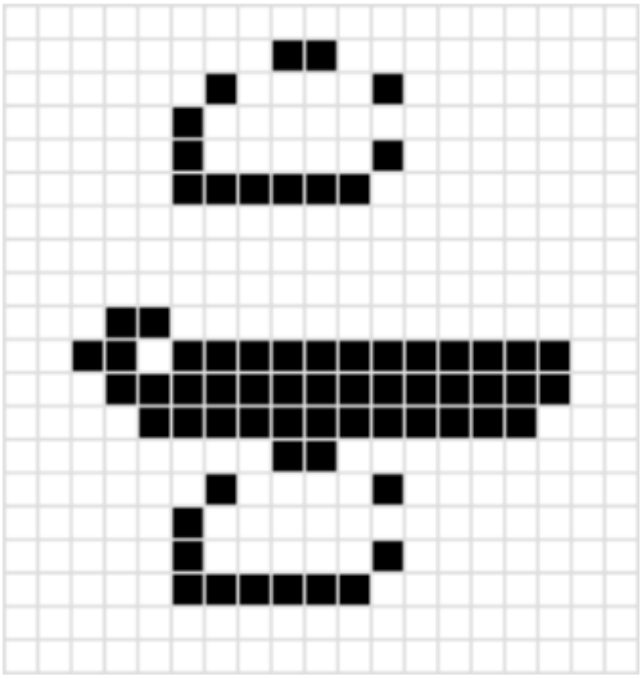


Figure 2.2.3.6: "Flotilla 1"

**Frothing spaceship:** is a spaceship whose back end appears to be unstable and breaking apart, but which nonetheless survives. The exhaust festers and clings to the back of the spaceship before breaking off.

**Droste spaceships:** also known as SMOS or spaceships consisting of multiple others spaceships colliding and fighting against each other.

To finalize this classification some images of "Gemini" are included, this is, a Droste-engineered self-constructing oblique spaceship created by Andrew J. Wade in 2010, which replaces 5120 cells vertically and 1024 cells horizontally every 33699586 generations. Therefore:

$$speed(Gemini) = \frac{(5120{,}1024) * c}{33699586}$$

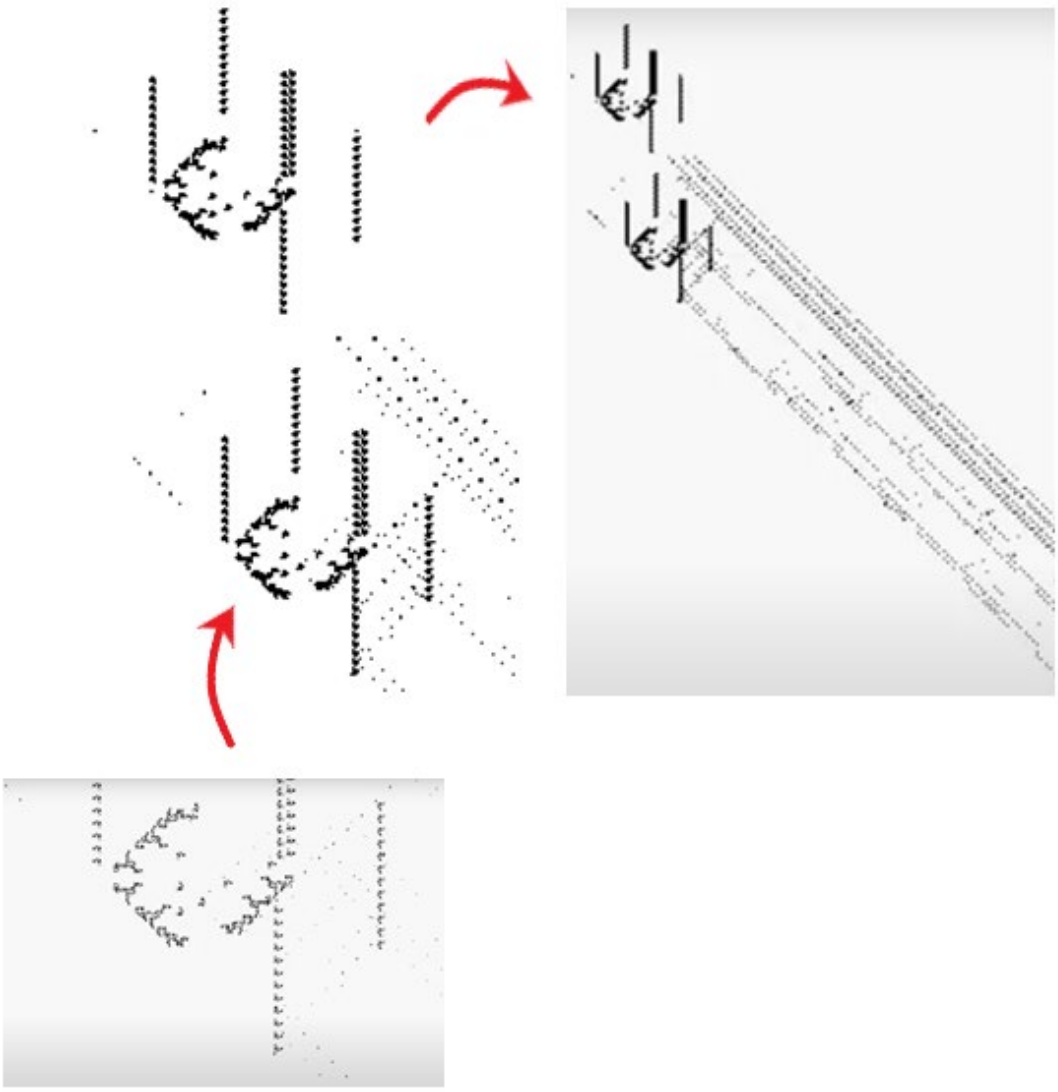


Figure 2.2.3.7: Some sections of “Gemini”

Given the impossibility of offering a static image that includes the entire pattern with sufficient resolution, I have chosen to show certain sections of the colossal Gemini.

Even so, as complex and infinite as the potential of this pattern may seem, the next unexpected structure in Conway’s Game of Life is unbelievable deeper and remains to be discussed. The pinnacle of complexity has not yet been reached.

### 2.2.4. *Methuselah (M) and other complex configurations*

Although it is true that every stable pattern and every oscillator can be considered infinite-growing patterns, they just accomplish the ultimate rule of living: survive, or at least not vanishing in Conway's Game of Life context. But the structures mentioned above are not able to periodically increase their population, which happens to be the key property in order to be considered an infinite pattern.

Historically, "Gosper Glider Gun" is the first known finite pattern with unbounded growth, discovered by Bill Gosper in November, 1970. Is also the first known glider generator (Gun), due to its particular and periodic creation of a Glider every 30 generations.

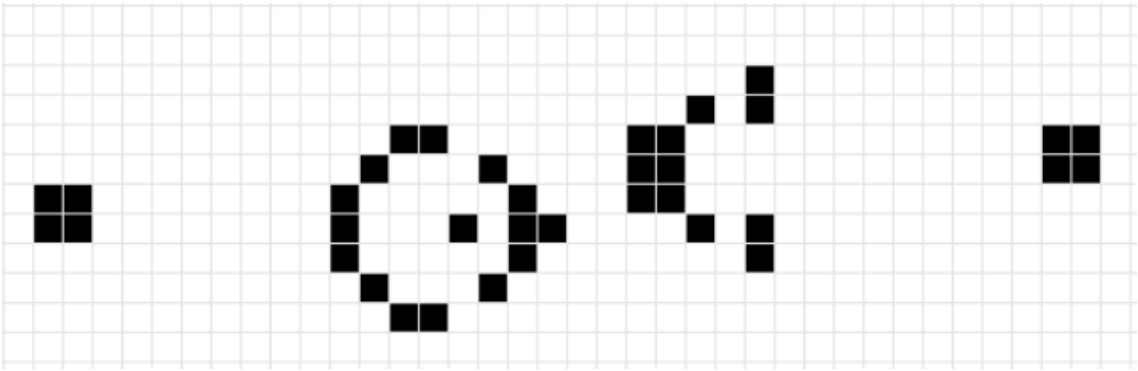


Figure 2.2.4.1: Initial configuration for "Gosper Glider Gun"

With this in mind, the following binary abstraction is considered: the presence of a flow of gliders is considered a 1, while its absence is a 0.

**Construction of the AND logic gate:** The figure 2.2.4.2 shows the initial setup required to create the door. To understand how it works, the structural pattern has been divided into several numbered sections.

- The natural behavior of the AND operation requires two binaries inputs, which can be found in sections (1) and (2).

- Section (9) represents the final output, but in order to abandon the logic gate bounding box provoking outside interactions, the glider flow provided by (8) must disappear. So its clear (8) is just acting as the output blocker, terminating its flow almost in every case, except when both inputs are active.

- Section (3) is the Rosebud in this system, the only way for shutting down the (8) flow, (triggering 1 as the final output) is to ensure that the flow of (3) reaches safe and sound to (8) through the entire glider framework.

- In order to evaluate if one input is active or not, the system need another Glider Gun acting as a sensor of proximity and this function is performed continuously by (4) and (5). Whenever it's associated input is active, both section (6) and our Rosebud (3) glider flows are correctly active too, owed to the sensors.

- And last but not least the section (7), also known as the Rosebud's nemesis or the final boss, which comes to represent the last obstacle between (3) and the output blocker. A constant flow that can never be stopped, but can be avoided if (3) changes its period by doubling it. An action that could not be performed without the necessary and coordinated help provided by (6) as it it shown in figure 2.2.4.3.

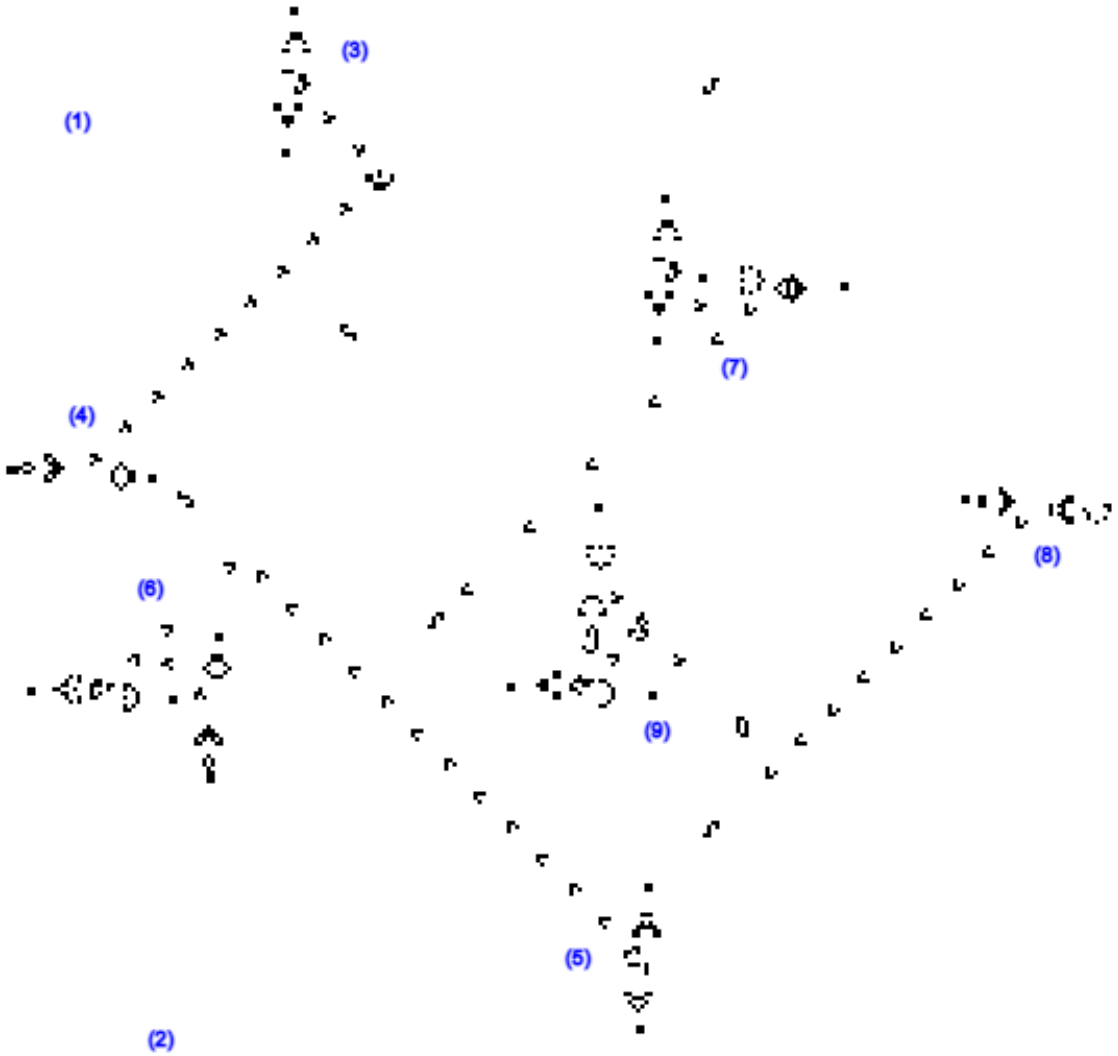



Figure 2.2.4.2: "AND logic gate" with both inputs deactivated

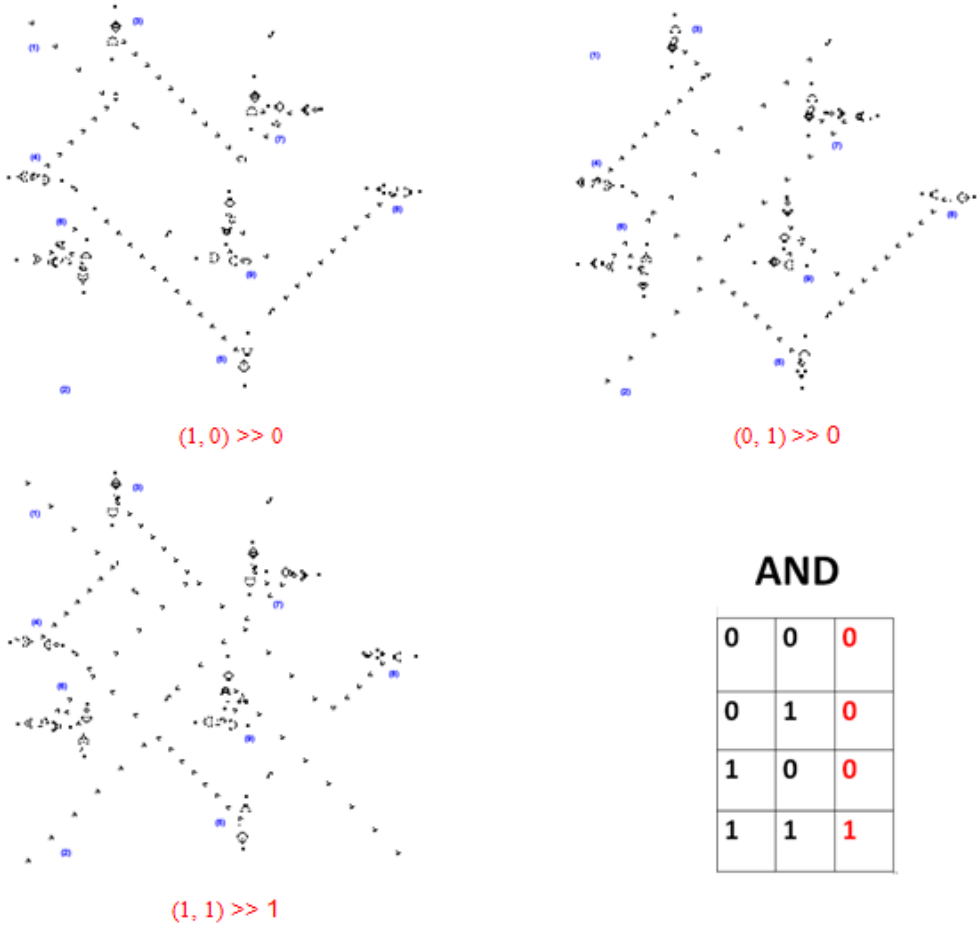



Figure 2.2.4.3: "AND logic gate" in all other cases and its Truth Table

**Methuselah (M):** is a pattern that take a large number of generations (known as lifespan) in order to stabilize but it ends up doing it against all odds. Often innocent looking, they are the most chaotic, mysterious, and infrequent patterns whose modus operandi is usually to abruptly expand its boundary box before suddenly stabilize. When it does so, the methuselah usually adopts a constellational configuration made up of various oscillators or still lifes, even producing spaceships along the way.

Given the huge amount of diverse methuselah discovered since the game of life inception and their disparate complexities, there is no consensus on the exact abstract definition nor the mathematical one either. Anyway, the unwritten rule of 100 generations is worldwide accepted, not considering a pattern as one methuselah until it stabilizes at more than that amount of generations.

On the other hand, it is needless to add that if vanishing is considered a stable pattern, then methuselah which eventually disappear are also considered in this classification as one subgroup known as "diehards". The smallest and most famous methuselah was

discovered by John Conway in 1969 and it's called "R-pentomino", which increases its population from 5 cells to 116 after stabilizing at 1103 generations and culminates its work generating 6 gliders along the way.

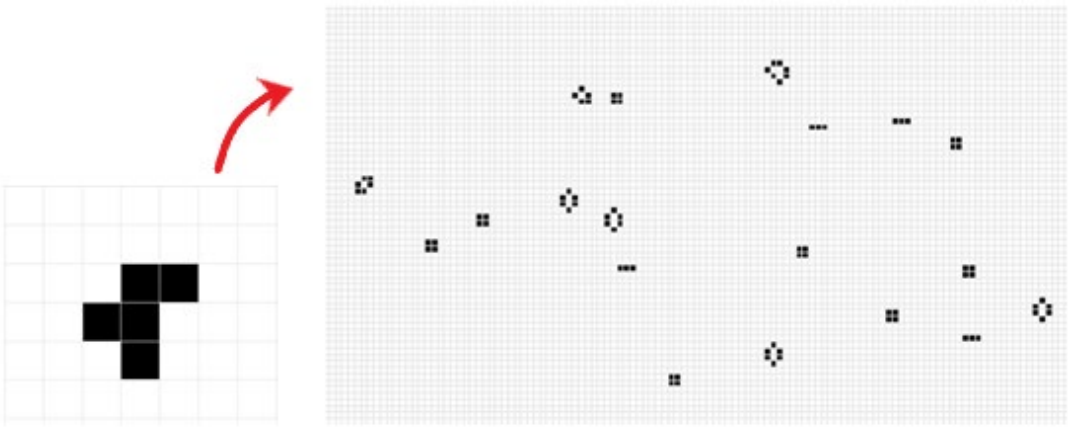


Figure 2.2.4.4: Initial and final configuration for "R-pentomino"

Throughout history, these patterns have generated great interest, often forming search communities and algorithms to find more, a trend that has continued to this day. One interesting example of this could be Soup searching, a popular method of finding methuselahs that fit within a given bounding box. Responsible of discovering "Fred" in 2010, the longest lasting soup found by Schneelocke using Nathaniel Johnston's search script which takes 35426 generations in order to stabilize.

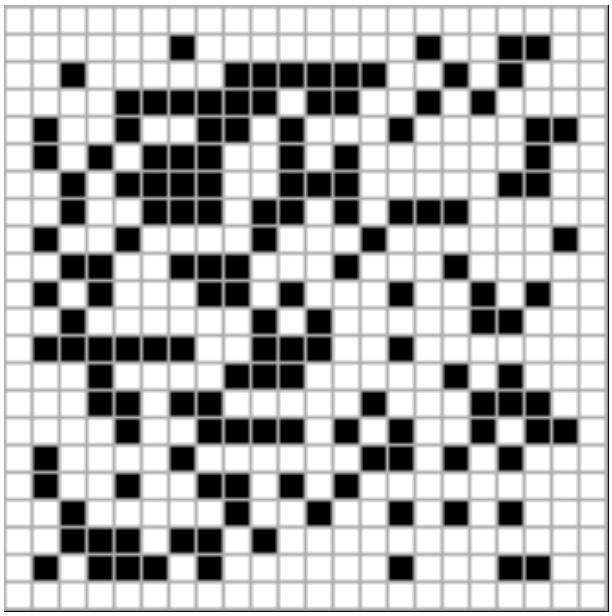


Figure 2.2.4.5: Initial configuration for "Fred"

## 2.3. **Running some initial basic configurations**

This section is dedicated to showing a light compilation of basic initial configurations. Despite its simplicity, in some cases certain really complex patterns are extracted. The idea is to find simply defined structures that statistically give rise to some Methuselahs.

The notation in the following tables is as follows:

- **NxM** refers to an initial rectangular configuration of N rows and M columns.

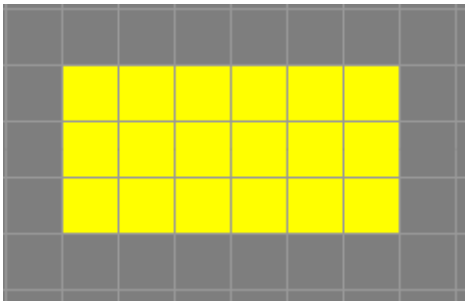


Figure 2.3.1: Initial “3x6” configuration

- **NxM d** refers to an initial configuration of N diagonals of M contiguous cells and arranged as shown in the two figures below.

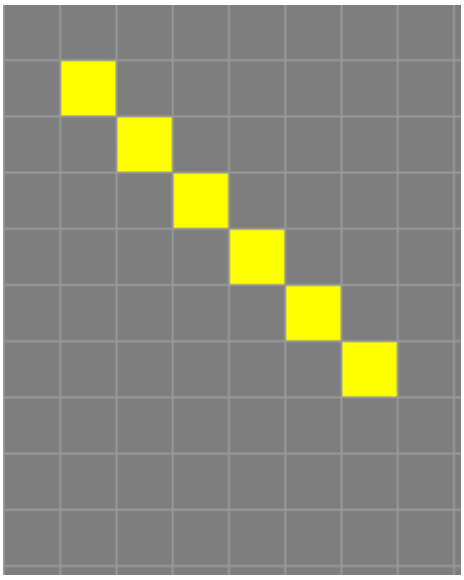


Figure 2.3.2: Initial “1x6 d” configuration

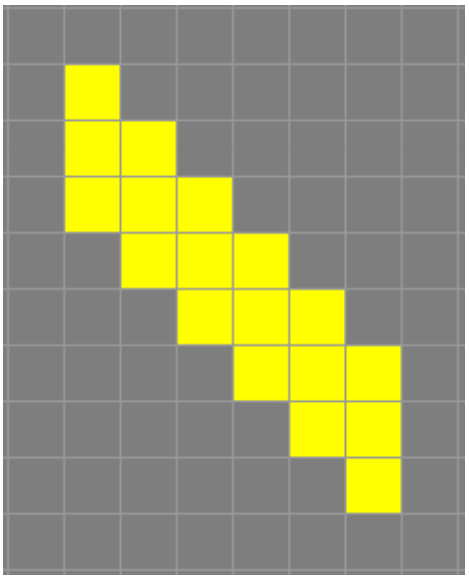


Figure 2.3.3: Initial “3x6 d” configuration

- As previously introduced: **S.L, O**, **S and M** refers to a Still life, oscillator, spaceship and methuselah respectively.

- From this point forward: **NxG.G** refers to a pattern that has generated N gliders at some point. While **C**, **V** are reserved for patterns based on constellations or void (vanishing structures).

It has been decided to limit NxM to a maximum of 7x16, with the idea of not over-expanding the boundary box of the initial configuration. Being N the rows and M the columns of the table, with strict destroverse and gravitational reading (from top to bottom and from left to right)

*NxM:*

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| V | V | O | S.L | 4xO | V | 4xS.L | 8xS.L | 8xO | O | 2xO | 2xS.L | 2xO | V | V | 8xO |
| V | S.L | S.L | S.L | V | V | V | V | 2xO | 2xS.L | 6xO | V | V | 6xS,L | 4xS.L | 4xS.L |
| O | S.L | 4xO | V | 4xS.L | 8xS.L | 8xO | O | 2xO | 2xS.L | 2xO | V | V | 8xO | 4xS.L | V |
| S.L | S.L | V | V | 8xS.L | S.L | S.L | S.L | V | 6xS.L | 8xO | V | 8xO | 4xS.L | V | V |
| 4xO | V | 4xS.L | 8xS.L | 4xS.L | 2xO | M+O | C | 2xO | C | 2xO | C | C | C | C | C |
| V | V | 8xS.L | S.L | 2xO | S.L | V | V | 8xO | V | 4xS.L | V | V | V | 2xO | 2xS.L |
| S.L | V | O | S.L | M+O | V | 4xO | 2xO | 10xO | 2xO | C | 2xO | 2xO | 2xO | 4xO | C |

Figure 2.3.4: Square configurations Table

If the objective of the section was to find abundant methuselahs, this first attempt could be considered a failure. This is because in these rectangular configurations stability shines by its presence. In the game of life, there are 90° rotation symmetry rules, causing isomorphism or invariance by changing rows and columns.

Even so, several initial configurations are found under resultant homeomorphism, and four patterns have been found that deserve to be mentioned:

- **Homeomorphism rule between orthogonal structures of one and three rows**:

Any initial configuration of the form 1 x M, after a generation, is transformed into a configuration of the form 3x (M-2).

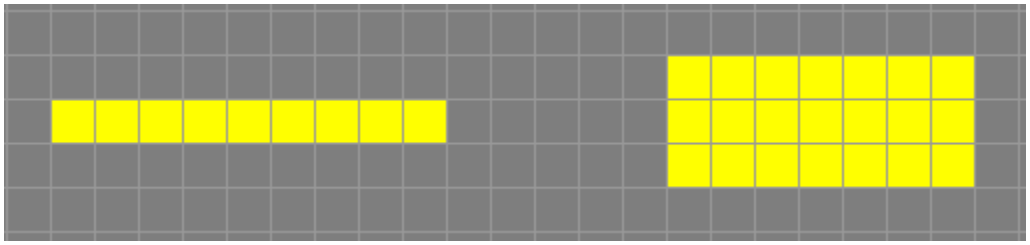


Figure 2.3.5: 1x9 and 3x7 under isomorphism

As a consequence of limiting the boundary box of the initial configurations, this resultant homeomorphism rule is the only one that can be found among the NxM configurations in this table. You can see the equivalence between rows 1 and 3 (shifted two columns to the right) in figure 2.3.4

- **Pentadecathlon Generators:**

The Pentadecathlon is a period-15 oscillator found by John Conway in 1970. It is the only known oscillator that is polyomino (a finite collection of orthogonally-connected cells) in more than one phase (besides the blinker), and it is the smallest oscillator with a period greater than its minimum population.

Figure 2.3.6: Pentadecathlon 15-period

As John Conway discovered, the orthogonal line of 10 cells evolves in this object, but thanks to the previously discussed homeomorphism rule, the initial 3x8 configuration also results in the same oscillator.

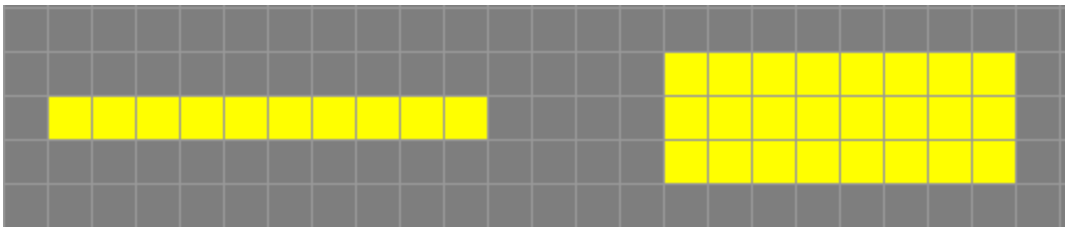


Figure 2.3.7: 1x10 and 3x8 as Pentadecathlon generators

- **Methuselahs found with these configurations have been:**

Being isomorphic structures by changing rows by columns, 5x7 and 7x5 evolve in the only methuselah found with the NxM table. After stabilizing in ≈140 generations, the methuselah results in a constellation made up of 4 still lives that enclose a period 3 oscillator.

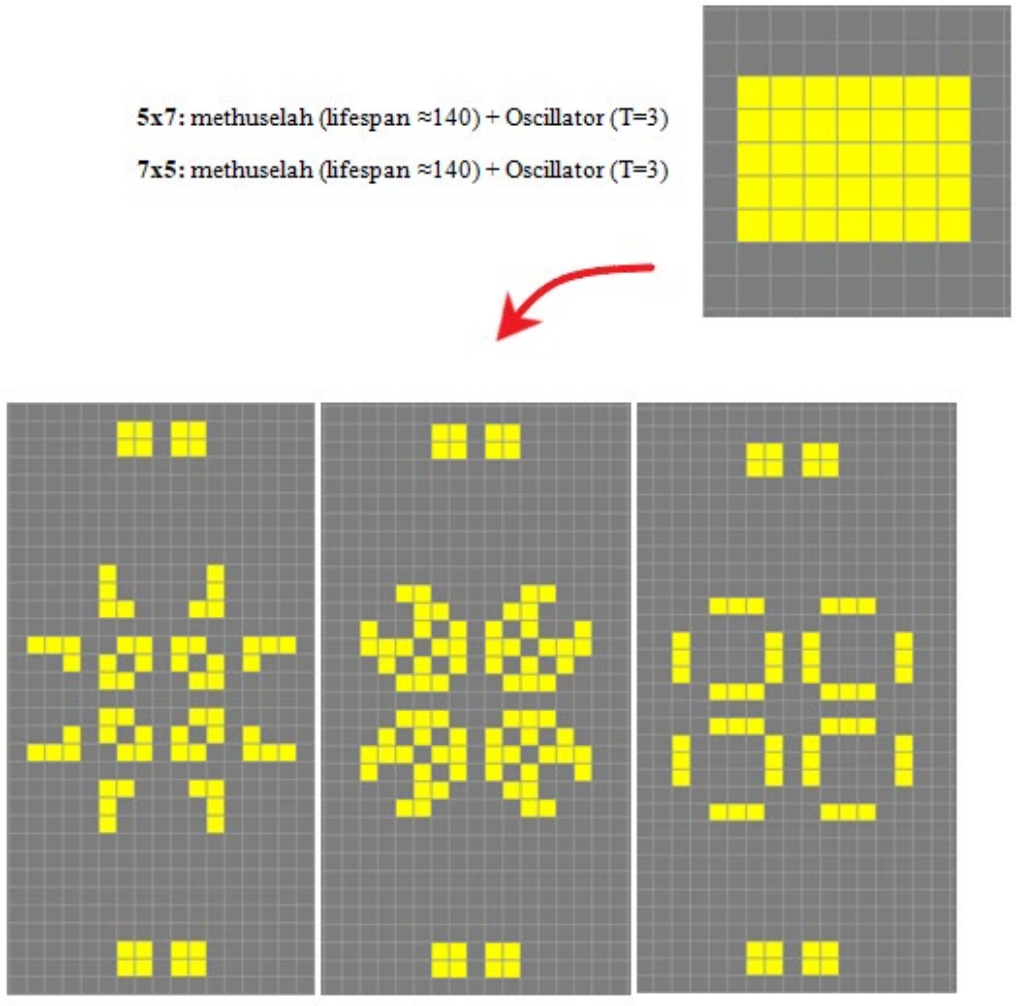



Figure 2.3.8: Found Methuselah in Squared Configurations

*NxM d:*

| V | V | V | V | V | V | V | V | V | V | V | V | V | V | V | V |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| V | S.L | 4xS.L | M+2xG.G | 4xS.L | 2xO | M | V | 2xS.L | V | V | 2xO | 2xO | M+4xG.G | 2xO | M |
| O | O | 4xS.L | O | V | 2xS.L | M | 2xGG | M+2xG.G | 4xG.G | 6xS.L | 6xS.L | 6xS.L | C | 6xS.L | 6xS.L |
| S.L | 4xS.L | 2xS.L | V | 2xS.L | 2xS.L | V | 2xO | V | C | 4xS.L | 2xS.L | 6xS.L | 3xS.L | 2xS.L | 2xS.L |
| O | 2xS.L | 2xS.L | M | 4xS.L | 2xS.L | V | M+2xG.G | M+2xG.G | C | M | 6xS.L | M+6xG.G | M | C | M+4xG.G |
| V | V | M+4xG.G | 2xS.L | 2xS.L | V | C | V | C | 2xS.L | C | M+6xG.G | M+2xG.G | 10xS.L | M+4xG.G | M+4xG.G |
| S.L | V | V | V | C | V | 4xO | M | V | V | 2xO+2GG | M+6xG.G | M+6xG.G | V | C | 8xS.L |

Figure 2.3.9: Diagonal configurations Table

Contrary to the previous table, the arrangement chosen this time has thrown enough randomness to generate several Methuselahs. By eliminating some typical Game of Life's isomorphism, the results have been more varied, exotic and have taken four times longer on average to stabilize than before.

The most frequent previous resulting pattern (the oscillator) has suddenly become the least obtained. More constellations and Methuselahs have appeared in addition to spaceships for the first time, specifically patterns that have generated various gliders.

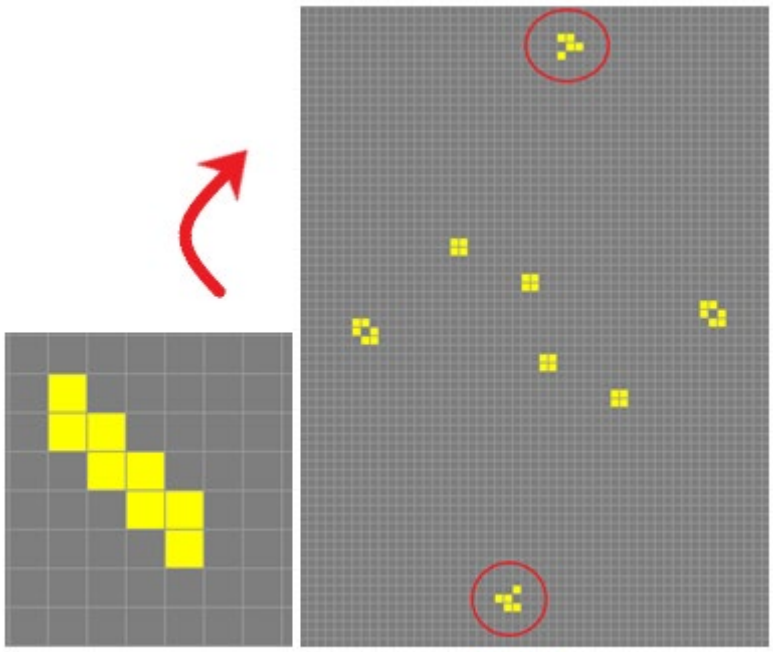


Figure 2.3.10: initial and final configuration for 2x4d

Gathering the last methuselahs obtained in figure 2.3.11, I conclude section 2 of the project. Having presented several previous concepts including the inception for abstract cellular automata, I hope Conway's game of life is satisfactorily explained. The moral that should remain is undoubtedly the zero need for complex rules and principles in order to provoke the genesis of beautiful and powerful results. Mathematics come to our rescue again.

- **Methuselahs found with these configurations have been:**

**2x4 d:** methuselah (lifespan≈117) + 2x Glider Generator

**2x7 d:** methuselah (lifespan≈640)

**2x14 d:** methuselah (lifespan≈640) + 4x Glider Generator

**2x16 d:** methuselah (lifespan≈213)

**3x7 d:** methuselah (lifespan≈124)

**3x9 d:** methuselah (lifespan≈397) + 2x Glider Generator

**3x18 d:** methuselah (lifespan≈180)

**5x4 d:** methuselah (lifespan≈397)

**5x8 d:** methuselah (lifespan≈342) + 2x Glider Generator

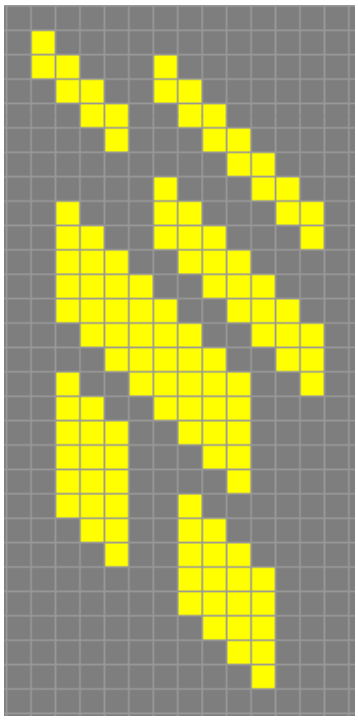
**5x9 d:** methuselah (lifespan≈280) + 2x Glider Generator

**5x11 d:** methuselah (lifespan≈120)

**5x13 d:** methuselah (lifespan≈624) + 6x Glider Generator

**5x14 d:** methuselah (lifespan≈168)

**5x16 d:** methuselah (lifespan≈1110) +4x Glider Generator



**6x3 d:** methuselah (lifespan≈217) +4x Glider Generator

**6x12 d:** methuselah (lifespan≈676) +6x Glider Generator

**6x13 d:** methuselah (lifespan≈187) +2x Glider Generator

**6x15 d:** methuselah (lifespan≈187) +4x Glider Generator

**6x16 d:** methuselah (lifespan≈187) +4x Glider Generator

**7x8 d:** methuselah (lifespan≈124)

**7x12 d:** methuselah (lifespan≈648) +6x Glider Generator

**7x13 d:** methuselah (lifespan≈205) +6x Glider Generator

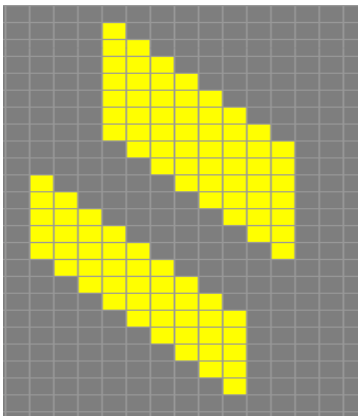


Figure 2.3.11: Found Methuselah in Diagonal Configurations

# 3. CONWAY'S GAME OF LIFE IMPLEMENTATION

## 3.1. Overview and Design decisions

All the content that can be programmed in the project has been done with Python. There are several reasons that pushed me to choose this language, but I should highlight two:

- It is a mainstream language right now. It has a large amount of documentation and libraries that can help me in both the design and coding phase. It is incredibly understandable and particularly efficient in certain operations typical of machine learning or artificial intelligence.

- As will be specified later in section 4 of the project, a very particular library will be used that gives the excuse to briefly comment on the transformers, one of the most spectacular advances in machine learning in recent years that almost took all the prominence of this project. As this library is written mostly in Python, for consistency and simplicity, my project also had to be done on Python and it can be found in the following GitHub repository:

github.com/Luis-Carles/yawnoc.git

The basic idea that I wanted to achieve was the correct distribution of functionalities. Classify all the functionality related to an abstract concept in an independent python file and achieve the correct collaboration between them. Since this document is also divided in different sections, from this point forward each python file will be explained in detail when necessary.

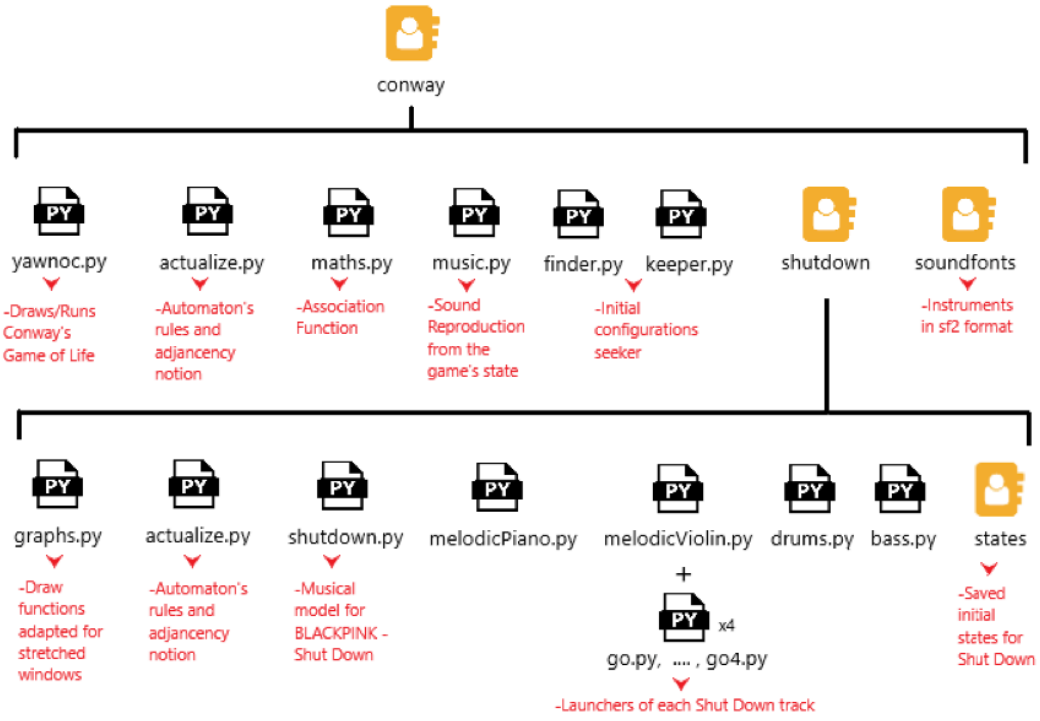



Figure 3.1.1: Distribution of project functionalities in different Python modules

The foreshadowed software product, from now on under the name yawnoc, will implement both the game of life and the music visualizer. It has been carried out respecting the guidelines of an agile methodology based on rapid work milestones until a relevant requirement is implemented. The 3.1 section concludes with the yawnoc requirements specification.

In the first place, the **functional requirements**:

| Requirement code | FR01 |
|---|---|
| Name | Game of life in static mode |
| Purpose | Show in a window a state of the game |
| Description | The application should offer the user a pop-up window showing the board of the game of life, initially empty. The dimensions of both the window and the board will be given by the user as a parameter. |
| Input | Application starts or blizzard action. |
| Output | A window and an initially empty board with the corresponding dimensions. |
| Priority | High |

| Requirement code | FR02 |
|---|---|
| Name | Random Universe |
| Purpose | Generates a random state for the game of life and declares it the next state. |
| Description | The application should offer the user the possibility to generate a random state of the game at any time, so that it is the next to show. |
| Input | Click on the "r" key in the usual keyboard input, performing a random state action. |
| Output | The next state of the game will be random. |
| Priority | Low |

| Requirement code | FR03 |
|---|---|
| Name | Playing God |
| Purpose | Create life or destroy it, changing the state of a cell on the board. |
| Description | The application should allow the user to alter the state of a cell of their choice whenever the game is in static mode. |
| Input | Left clicking on a cell on the screen, performing an alter action. |
| Output | The chosen cell will be alive (1) if it was dead before (0) and vice versa. |
| Priority | Medium |

| Requirement code | FR04 |
|---|---|
| Name | Gridding the Board |
| Purpose | Create a grid on the board. |
| Description | The application should allow the user to decide if he wants a grid board or not. In order to better distinguish the limits of each cell whenever the game is in static mode. |
| Input | Click on the "g" key in the usual keyboard input. |
| Output | When the next state is rendered, the board will have grid if not before and vice versa. |
| Priority | Low |

| Requirement code | FR05 |
|---|---|
| Name | Clear Universe |
| Purpose | Return to the initial state of the game of life, with an empty state. |
| Description | The app should allow the user the ability to clear the board, forcing the game back to its initial state with an empty universe. |
| Input | Click on the "c" key in the usual keyboard input. |
| Output | The next state will have all its cells dead (0) |
| Priority | Low |

| Requirement code | FR06 |
|---|---|
| Name | Next State |
| Purpose | Advance to the next state of the game of life. |
| Description | The application should allow the user to advance to the next state, following Conway's local game of life rule set, and only if the game is in static mode. |
| Input | Clicking the right arrow pad on the standard keyboard input. |
| Output | The next state of the game of life is calculated and displayed on the board. |
| Priority | Medium |

| Requirement code | FR07 |
|---|---|
| Name | Game of life in dynamic mode |
| Purpose | The following states of the game of life are successively displayed on the screen. |
| Description | The application should allow the user to enter dynamic mode, where the next state of the game is calculated and displayed automatically at a given speed. |
| Input | Clicking the space bar of the standard keyboard input. |
| Output | The dynamic mode is activated, entering a constant flow of status updates. |
| Priority | High |

| Requirement code | FR08 |
|---|---|
| Name | Speed Management |
| Purpose | Slow down or speed up the game of life. |
| Description | The application should allow the user to control the rate of updating to the next state of the game of life, whenever the game is in dynamic mode. |
| Input | Clicking the Up/Down arrow pad on the standard keyboard input. |
| Output | Constant flow speed increases/decreases as desired by the user. |
| Priority | Medium |

| Requirement code | FR09 |
|---|---|
| Name | Blizzard |
| Purpose | Freeze the game of life. |
| Description | The application should allow the user to freeze the game of life and return to static mode if not already in it. |
| Input | Clicking the space bar of the standard keyboard input in dynamic mode. |
| Output | The game stops again in static mode, showing the last calculated state on the screen. |
| Priority | Low |

| Requirement code | FR10 |
|---|---|
| Name | Musical Approach |
| Purpose | Reproduction of the sound associated with each state. |
| Description | Whether in static or dynamic mode, the application should allow the user to hear a sound associated with each state of the game of life. This sound will be reproduced by a piano/violin/drums/bass at the user's choice and will be calculated based on the arrangement of its cells. |
| Input | Clicking the space bar of the standard keyboard input. |
| Output | Application starts in a no-casual mode. |
| Priority | High |

| Requirement code | FR11 |
|---|---|
| Name | Visualizer |
| Purpose | Model a particular song on Conway's Game of Life. |
| Description | The application should allow the user to search for melodic/rhythmic/harmonic patterns of their choice to model a song with the game of life. |
| Input | The application is started in pattern search mode. |
| Output | An unguaranteed search for the pattern indicated by the user is initiated. |
| Priority | Medium |

| Requirement code | FR12 |
|---|---|
| Name | BLACKPINK Shut Down |
| Purpose | Modeling BLACKPINK - Shut Down's tracks in the game of life. |
| Description | The application should allow the user to listen to an example model present in the application, corresponding to the four tracks of said piece simultaneously on the screen. |
| Input | The application is started in example mode. |
| Output | The Shut Down model begins. |
| Priority | High |

And finally, the **non-functional requirements**:

| Requirement code | NFR01 |
|---|---|
| Name | Look and feel |
| Description | The appearance of the interface must be always consistent, as well as being intuitive, accessible, and easy for the user to learn. If multiple pop-up windows appear at the same time, in no case should individual functionality be affected. |
| Priority | Medium |

| Requirement code | NFR02 |
|---|---|
| Name | Separation of powers |
| Description | The functionality of the application must be correctly distributed in different modules, allowing the user to easily make changes and imports where appropriate. |
| Priority | Low |

## 3.2. **yawnoc.py**

Sinistroversely, Conway, is the python module in charge of the basic execution and the graphing of Conway's game of life. Leaning on captures, the file begins with:

```
1   import actualize as a
2   import music as m
3   import sys
4   from OpenGL.GL import *
5   from OpenGL.GLU import *
6   from OpenGL.GLUT import *
7   import time
8   import random
9
```

The first two `import` statements are executed to bring together the functionality of actualize.py and music.py, two other modules of the project. The rest of the lines reflect the libraries used in this file:

- **SYS**: Standard library for input-parameters management.

- **OpenGL:** Graphical interface chosen to represent the game of life in 2D. Particularly used in [3] and generally in many programs and video games, such as *Minecraft* or *Blender*.

- **Time**: Standard library to make pauses and control certain actions that require a specific pulse or tempo.

- **Random:** Standard library for generating pseudorandom arrangements.

```
10  ###Parameters Input Control
11
12  if len(sys.argv) != 4:
13      print("\nERROR: wrong parameters, usage: \n\npython yawnoc.py"
14      + "\n\t[musical Approach] "
15      + "\n\t[window's Dimension] "
16      + "\n\t[universe's Dimension]")
17      exit(-1)
```

That previous code is a sentence that prematurely terminates the execution of the program if exactly 3 input parameters have not been given. Also, it allows me to explain how yawnoc.py should be executed through the Anaconda prompt:

```
>python yawnoc.py [musical Approach][window's Dimension][universe's Dimension]
```

Figure 3.2.1: Execution of yawnoc.py in Anaconda Prompt

python yawnoc.py must be preceded by 3 non-negative integer input parameters.

- **[musical Approach]:** It refers to the way of approaching sound reproduction according to the state of the game of life, feature that will be explained later. For now, it is enough to know that it must be an integer between 0-5 and that it works as follows:

```
19  if (int(sys.argv[1])<0) or (int(sys.argv[1])>5):
20      print("\nERROR: wrong musical approach, \n\nmust use a"
21      +"\n\tnumber between 0-5 for:"
22      +"\n\t[musical approach]"
23      + "\n\t0: Casual (No Music)"
24      + "\n\t1: Melodic Piano"
25      + "\n\t2: Armonic Piano"
26      + "\n\t3: Rhytmic & Drums"
27      + "\n\t4: Melodic Violin"
28      + "\n\t5: Melodic Bass")
29      exit(-1)
```

- **[window's Dimension]:** it refers to the size, in pixels, dedicated to the popup that OpenGL will generate with the game of life. It must be a multiple of 10.

- **[universe's Dimension]:** it refers to the size, in # number of cells, dedicated to the matrix of integers that will give rise to the universe of the game of life. It must be a multiple of 10.

```
31  if ((int(sys.argv[2])%10)!=0) or ((int(sys.argv[3])%10)!=0):
32      print("\nERROR: wrong dimensions, \n\nmust use a"
33      +"\n\tmultiple of 10 for:"
34      +"\n\t[window's Dimension] "
35      +"\n\t[universe's Dimension]")
36      exit(-1)
```

For example, as indicated in the readme.md, an execution of:

```
>python yawnoc.py 1 800 100
```

Figure 3.2.2: Running example of yawnoc.py in Anaconda Prompt

will start the game of life in an 800x800 window, using a 100x100 cell matrix and playing an associated musical note through a piano in each state of the game.

Afterwards, the program continues with the global variables and making some declarations:

```
40  ### Initial declarations and global variables
41
42  speed= 0.8          #Standard speed for music approaches
43  stop = True
44  paintGrid = True
45
46  mApproach = int(sys.argv[1])
47
48  if(mApproach==0):  #Standard speed for casual approach
49      speed = 0.1
50
51  # Window dimensions
52  dimXwindow = int(sys.argv[2])    #standard: 800
53  dimYwindow = int(sys.argv[2])
54
55  # Universe dimensions
56  dimXgrid = int(sys.argv[3])      #standard: 100
57  dimYgrid = int(sys.argv[3])
58
59  # Cell dimensions
60  cellSize = dimYwindow / dimYgrid
```

All of them are related to basic aspects of the game: the flow's speed in state changing, the cell size in the window, a control parameter that determines if the game is stopped or not, and another that determines if the program should paint the grid in the background. This code section ends with the RGB colors reserved for the game, which vary according to the chosen musical approach.

```
62  # Colors
63  ORANGE = [1.0, 0.50, 1.0]
64  BLUE = [0.0, 0.0, 1.0]
65  GREEN = [0.0, 1.0, 0.0]
66  RED = [1.0, 0.0, 0.0]
67  YELLOW = [1.0, 1.0, 0.0]
68  BLACK = [0.0, 0.0, 0.0]
69  WHITE = [1.0, 1.0, 1.0]
70  GRAY = [0.15, 0.15, 0.15]
71  PINK = [1.0, 0.0, 1.0]
72  PURPLE = [0.5, 0.0, 1.0]
73  BROWN = [0.50,0.25,0]
74
75  winColor = BLACK
76  gridColor = GRAY
77
78  if(mApproach==1) or (mApproach==0) : #Melodic Piano Approach / Casual
79      cellColor = PINK
80
81  if(mApproach==2):     #Armonic Approach
82      cellColor = ORANGE
83
84  if(mApproach==3):     #Rhytmic Approach
85      cellColor = BLUE
86
87  if(mApproach==4):     #Melodic Violin Approach
88      cellColor = WHITE
89
90  if(mApproach==5):     #Bass Approach
91      cellColor = BROWN
```

At this point the environment is ready, therefore, thanks to the good conditions, the big bang arises. The matrix of cells is created and begins the function declaration:

```
95   ### Big Bang
96
97   universe = [[ 0  for i in range(dimXgrid)] for j in range(dimYgrid)]
98
99   # Gridding the universe
100  def grid():
101      global paintGrid
102
103      glColor3fv(gridColor)
104      glLineWidth(0.001)
105      glBegin(GL_LINES)
106
107      xFrac = dimXwindow / dimXgrid
108      yFrac = - dimYwindow / dimYgrid
109
110      i = 0
111      while i <= dimXwindow:
112          glVertex2f(i,0)
113          glVertex2f(i,-dimYwindow)
114          i += xFrac
115
116      j = 0
117      while j >= -dimYwindow:
118          glVertex2f(0,j)
119          glVertex2f(dimXwindow,j)
120          j += yFrac
121
122      glEnd()
```

The function responsible for drawing the grid in the background is `grid()`. It needs to import the global boolean variable that indicates whether to paint or not, then iterate through the entire array with two while loops. It uses calls to OpenGL's `glVertex()` to draw the lines.

```
124  # Creating the window
125  def init():
126      global winColor
127      glClearColor(winColor[0], winColor[1], winColor[2], 1)
128      glMatrixMode(GL_PROJECTION)
129      gluOrtho2D( 0, dimXwindow, -dimYwindow, 0 )
```

Despite having created the universe we still couldn't see it, that is where the `init()` function creates the popup window that will show the game of life. The reason why the Y dimensions of the screen are entered negatively deserves special mention.

This is because the OpenGL coordinates are reversed with respect to the popup window. The initial value (0,0) would be in the lower left corner of the screen, but OpenGL would place it in the upper left corner. From here on out, this will be solved by reversing the Y coordinates in `gluOrtho2D()` for consistency and coherence.

The code continues with the functions related to drawing the cells:

```
133   ###  Drawing actions
134
135   # Cell's position
136   def getCell(posX, posY):
137       x = int(posX / (dimXwindow / dimXgrid))
138       y = int(posY / (dimYwindow / dimYgrid))
139       return x,y
140
141   # Inverts the y axis maintaining consistency with openGL behaviour
142   def getCellonScreen(xCell, yCell):
143       x = int(xCell * (dimXwindow / dimXgrid))
144       y = -1 * int(yCell * (dimYwindow / dimYgrid))
145       return x,y
146
147   def drawSquare(xPos, yPos):
148       global cellColor
149
150       glColor3fv(cellColor)
151       glBegin(GL_POLYGON)
152       glVertex2f(xPos, yPos)
153       glVertex2f(xPos + cellSize, yPos)
154       glVertex2f(xPos + cellSize, yPos - cellSize)
155       glVertex2f(xPos, yPos - cellSize)
156       glEnd()
```

- **getCell():** returns the coordinates of a cell in the matrix.
- **getCellonScreen():** returns the coordinates of a cell in the game window.

- **drawSquare():** it is self-definable by its name, given some coordinates it draws a cell in the universe. The color is given by the global variable, while it creates the four vertices of the square as follows:

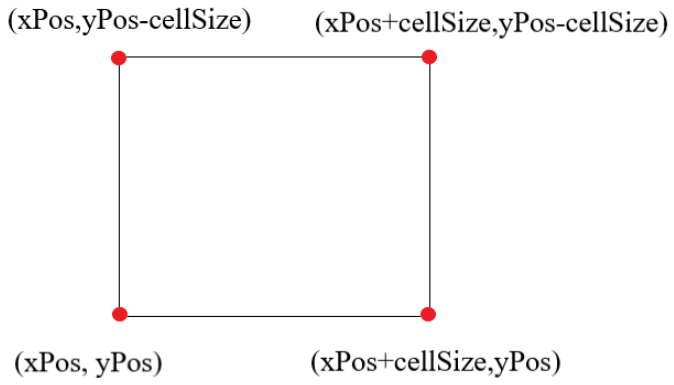



Figure 3.2.3: Four vertices of the square drawn by drawSquare()

```
158    # Draw alive cells in the universe
159    def drawLivings():
160        for j in range(dimYgrid):
161            for i in range(dimXgrid):
162                if universe[j][i] == 1:
163                    x,y = getCellonScreen(i,j)
164                    drawSquare(x,y)
165        glFlush()
```

- **drawLivings():** function that loops through the array, calling `drawSquare()` only if the cell is alive (1).

By continuing to the next section, runtime actions can be found. That is, actions that the user can perform while the execution of the program is active, in order to control Conway's game of life. The first are the basic keyboard actions:

```
170  ### Runtime actions:
171
172  def standardKeyboard(key, x, y):
173      global stop
174      global paintGrid
175      global universe
176      global winColor
177      global cellColor
178
179      # Clear "c"
180      if key == b'c' or key == b'C':
181          universe = [[ 0  for i in range(dimXgrid)] for j in range(dimYgrid)]
182
183      # Draw/Undraw the grid "g"
184      elif key == b'g':
185          paintGrid = not paintGrid
186
187      # Blizzard " "
188      elif key == b' ':
189          stop = not stop
190
191      # Randomize "r"
192      elif key == b'r' or key == b'R':
193          universe = [[ random.choice([0,0,0,1,1,1]) for i in range(dimXgrid)] for j in range(dimYgrid)]
```

- **“C” Key:** exterminate (0) all the cells in the universe, making a clear.

- **“G” Key:** decides whether to draw or not the grid in the next state.

- **“B” Key:** decide whether to stop or leave the automatic execution of the game of life, making a blizzard. While the game is stopped, a series of additional actions can be carried out.

- **“R” Key:** the next state of the universe will be random, making chaos. The cells that will arise will be 3 out of 6, a perfectly adaptable parameter to reduce randomness.

Followed by the unique standard-mouse and additional keyboard speed-dedicated runtime actions:

```
196 def aditionalKeyboard(key, x, y):
197     global speed
198     global universe
199 
200     # Next state  "⇨"
201     if key == GLUT_KEY_RIGHT and stop:
202         universe = a.nextState(universe)
203 
204     # Accelerate "⇧"
205     if key == GLUT_KEY_UP:
206         speed -= speed * 0.5
207 
208     # Decelerate "⇩"
209     if key == GLUT_KEY_DOWN:
210         speed += speed * 0.5
211 
212     glutPostRedisplay()
```

```
214 def standardMouse(button, state, x, y):
215     global universe
216     global stop
217 
218     # Playing God: creating life or destroying it
219     if button == GLUT_LEFT_BUTTON and state == GLUT_DOWN :
220         xCell, yCell = getCell(x, y)
221         previousState = universe[yCell][xCell]
222         universe[yCell][xCell] = 1 if previousState == 0 else 0
```

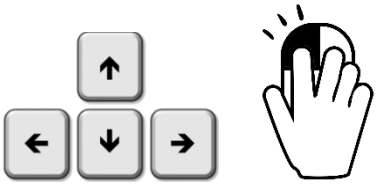


Figure 3.2.4: Standard abstraction for arrow pad and right click

- **“Right” direction key:** if the game is stopped, it allows a single update to the next state.

- **“Up” direction key:** increases the speed of the game of life, accelerating each change of state.

- **“Down” direction key:** decreases the speed of the game of life, slowing down each change of state.

- **“Left Click”:** if the game is stopped, it allows changing the state of a single cell of the matrix.

Python file yawnoc.py ends with the logical structure that supports the dynamic reproduction of the game of life and its visual representation through OpenGL. The idea is to gather all the game's own actions in the `main()` function, and then create an OpenGL loop that performs both the `main()` function and the relevant graphics and runtime actions.

```
226  ### Main body of the game of life's loopeable execution
227
228  def main():
229      global universe
230      global stop
231
232      glClear(GL_COLOR_BUFFER_BIT)
233
234      if paintGrid:
235          grid()
236
237      drawLivings()
238
239      # If music is on
240      if(mApproach!=0):
241       m.music(universe,mApproach,speed)
242
243      if not stop:
244          if(mApproach==0):
245           time.sleep(speed)
246
247          universe = a.nextState(universe)
248          glutPostRedisplay()
```

First, the universe and the global variable stop are added. The loop for each execution would be:

- Painting the grid if necessary.
- Draw living cells of the new state.
- If a musical approach was chosen, carry out the additional actions of music.py.
- If the game is not stopped dynamic mode is started, where the states are updated automatically.

As an additional comment, the speed of the flow is managed with the time.sleep sentence on line 245. But this is so if the game is in casual mode (no musical approach selected), in case one `mApproach` is chosen, the `time.sleep` statement will be executed during the execution of music.py.

The last section of the code assembles the OpenGL loop:

```
250   # OpenGL actions carried once
251   glutInit(sys.argv)
252   glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB)
253   glutInitWindowPosition(400,10)
254   glutInitWindowSize(dimXwindow,dimYwindow)
255   glutCreateWindow(b'Conways Game of Life')
256
257   init()
258
259   # OpenGL loop for main() function and runtime actions
260   glutDisplayFunc(main)
261   glutKeyboardFunc(standardKeyboard)
262   glutSpecialFunc(aditionalKeyboard)
263   glutMouseFunc(standardMouse)
264   glutMainLoop()
```

Thanks to the comments, the division between actions that are only executed once and those that enter the loop is clearly observed:

**Unitary execution:**

- Input parameters management.
- Creation of the popup window with the dimensions provided and in a centered position.
- Call to the `init()` function that adds the universe's matrix to the popup window.

**Iterated execution:**

- Conway's game-of-life actions previously gathered in main().
- Runtime actions of the standard/additional keyboard and the standard mouse.

## 3.3. **actualize.py**

It is the Python module of this project in charge of coding the internal logic of Conway's game of life. It contains the rules that determine the update of the value of the variable of each cell of the tessellated automaton and the concept of neighborhood on which these rules are based. Compartmentalization would make it possible to vary the rules of the cellular automaton at any time. Not restricting the project to the game of life and only forcing to change this python file without altering the rest.

It is recalled that the neighborhood concept on which the game of life works is Range-1 Moore's neighborhood. It does not seem excessively redundant to me to recover the figure already shown previously:

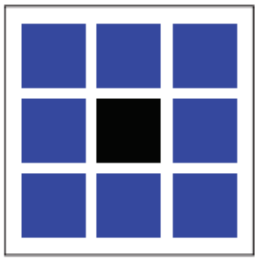


Figure 3.3.1: Range-1 Moore's Neighborhood

The design problem that immediately arises when dealing with this neighborhood is the dilemma between infinite and finite. An infinite universe without limits is assumed in the game of life. However, this is impossible within current computing. Until now the size of the universe has been limited to offer a faster and more efficient game of life. This implies considering that in these extremes, evaluating the neighborhood would sometimes throw exceptions for exceeding the matrix boundaries.

While the problem could have been solved by using the `try` and `catch` sentences to handle any exceptions that might occur in actualize.py, I decided to classify the cells into several classes and treat each one separately:

- **Upper Left Corner** (orange)
- **Upper Limit** (black)
- **Upper Right Corner** (green)
- **Lower Left Corner** (purple)
- **Lower Limit** (rose)
- **Lower Right Corner** (blue)
- **Western Limit** (yellow)
- **Eastern Limit** (brown)
- **Any other cell** (red)

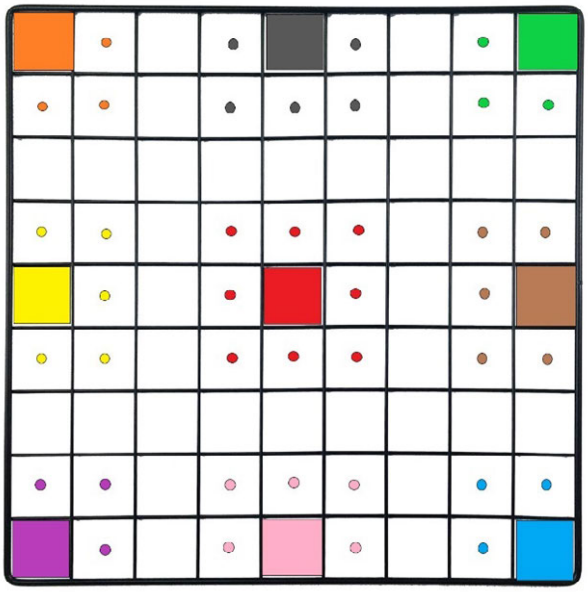


Figure 3.3.2: Cells Classification to resolve boundaries problem

The coding for the Moore neighborhood, considering the previous division, looks like this:

```
### Moore Neighborhood

def neighborhood(universe, x, y):
    n = 0
    if(y==0) and (x==0):        #upper left corner
        n=n+universe[y][x+1]
        n=n+universe[y+1][x]
        n=n+universe[y+1][x+1]

    if(y==0) and (x==len(universe[0])-1):    #upper right corner
        n=n+universe[y][x-1]
        n=n+universe[y+1][x-1]
        n=n+universe[y+1][x]

    if(y==0) and (0<x) and (x<len(universe[0])-1):    #upper limit
        n=n+universe[y][x-1]
        n=n+universe[y][x+1]
        n=n+universe[y+1][x-1]
        n=n+universe[y+1][x]
        n=n+universe[y+1][x+1]

    if(y==len(universe)-1) and (x==0):     #lower left corner
        n=n+universe[y][x+1]
        n=n+universe[y-1][x]
        n=n+universe[y-1][x+1]
```

where some borderline cases are treated only by checking the cells in the neighborhood that will not cause problems.

```
    if(y==len(universe)-1) and (x==len(universe[0])-1):     #lower right corner
        n=n+universe[y][x-1]
        n=n+universe[y-1][x-1]
        n=n+universe[y-1][x]

    if(y==len(universe)-1) and (0<x) and(x<len(universe[0])-1):   #lower limit
        n=n+universe[y][x-1]
        n=n+universe[y][x+1]
        n=n+universe[y-1][x-1]
        n=n+universe[y-1][x]
        n=n+universe[y-1][x+1]

    if(0<y<len(universe)-1) and (x==0):     #western limit
        n=n+universe[y][x+1]
        n=n+universe[y-1][x]
        n=n+universe[y-1][x+1]
        n=n+universe[y+1][x]
        n=n+universe[y+1][x+1]

    if(0<y) and (y<len(universe)-1) and (x==len(universe[0])-1):     #eastern limit
        n=n+universe[y][x-1]
        n=n+universe[y-1][x]
        n=n+universe[y-1][x-1]
        n=n+universe[y+1][x]
        n=n+universe[y+1][x-1]
```

Finally, if we are faced with the least restrictive case, where all neighboring cells can be evaluated without problem, a total of 8 verifications can be displayed:

```
    if(0<y) and (y<len(universe)-1) and (0<x) and (x<len(universe[0])-1):     #any other cell
        n=n+universe[y][x-1]
        n=n+universe[y][x+1]
        n=n+universe[y-1][x-1]
        n=n+universe[y-1][x]
        n=n+universe[y-1][x+1]
        n=n+universe[y+1][x-1]
        n=n+universe[y+1][x]
        n=n+universe[y+1][x+1]

    return n
```

The *n*-value that the `neighborhood()` function returns, is the count of all the living cells that the cell has around. It is a crucial data, according to these rules, to determine the next value of the variable relative to each cell.

The last fragment of actualize.py contains the rules of the game or of life. However, only the rules that allow a cell to live or be reborn are those that have been contemplated. This is so because the next state initially goes through a clear operation, therefore it would suffice to mark the living cells of the next state. Those that would die are already dead without further actions.

```python
### Game of life Rules

def nextState(universe):

    height = len(universe)
    width = len(universe[0])

    nextUniverse = [[0] * width for _ in range(height)]

    #calculating new state
    for y in range(height):
        for x in range(width):
                if(universe[y][x]==0) and (neighborhood(universe,x,y)==3): #Dead Cell Reborn with 3 living cells
                        nextUniverse[y][x]=1
                if(universe[y][x]==1) and (neighborhood(universe,x,y)==2): #Living cell remains alive with 2 living cells
                        nextUniverse[y][x]=1
                if(universe[y][x]==1) and (neighborhood(universe,x,y)==3): #Living cell remains alive with 3 living cells
                        nextUniverse[y][x]=1


    return nextUniverse
```

To conclude section 3 of the project, dedicated to presenting the coding of the original game of life without additional inclusions, it seems to me visually beautiful to include a capture of a random execution in casual mode some states after randomizing the universe.

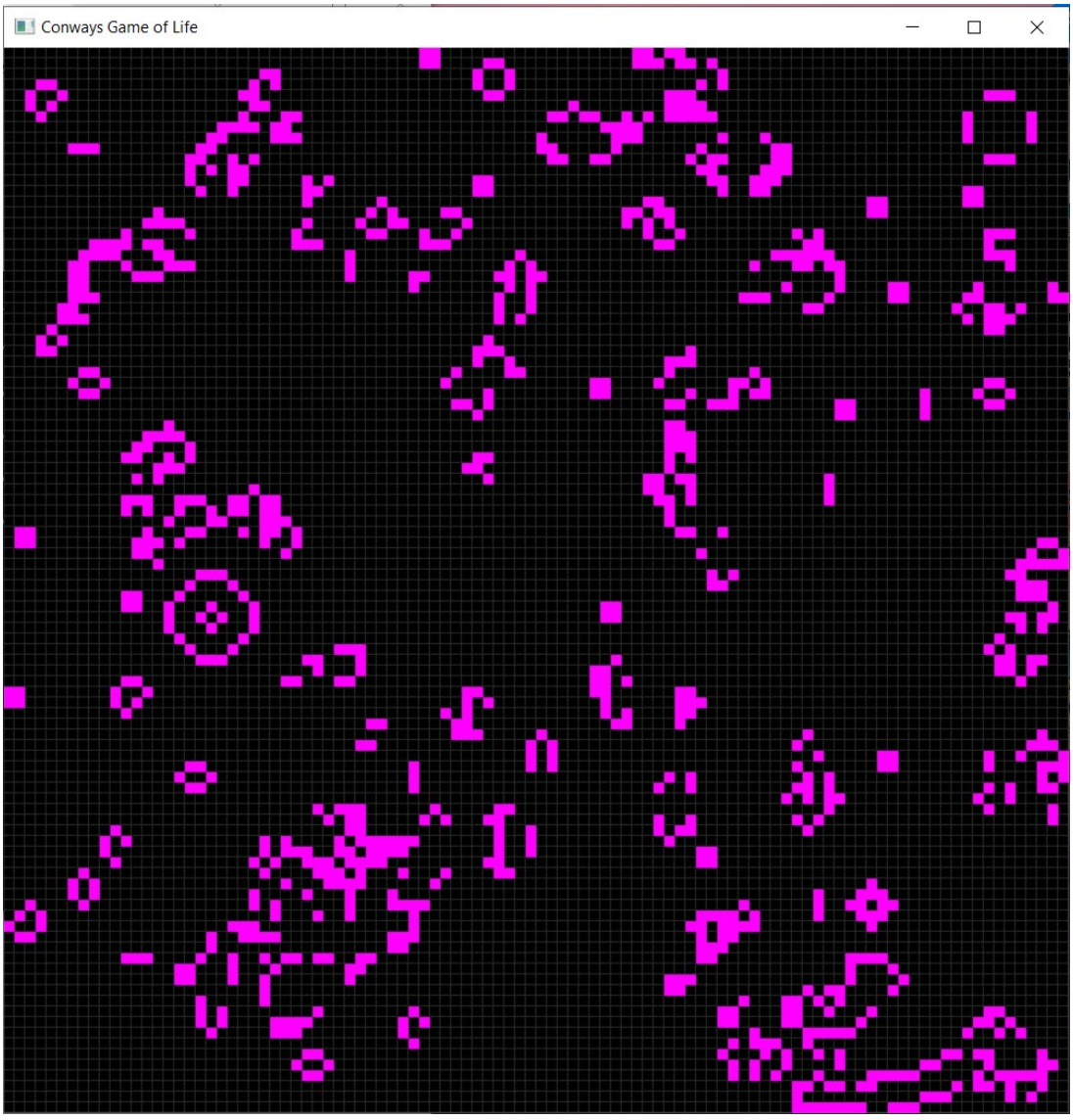



Figure 3.2.5: Capture of "python yawnoc.py 0 800 100" after pressing "R" key

# 4. SOUND AND MUSIC WITHIN CONWAY'S GAME OF LIFE

## 4.1. **maths.py: association function between grid and sound**

It could be said that the section that nobody asked for except myself has arrived. I often struggle with the feeling that my desires and aspirations do not correspond to the majority, but lately instead of seeing it as a curse I have started to see it as a miracle.

During the brainstorming that preceded the planning of this project, the idea of studying visualizers came up, a new trend that helps the hearing-impaired community by creating a graphical representation of music. This idea coexisted on a final podium, occupied by the game of life and transformers, and was about to be discarded for not finding a direct application of it. That was when I had the epiphany. I had a game that ultimately generated very suggestive mosaics and the intention to create music through images. When I made the decision and the idea went viral and took root in my mind, the first question I asked myself was: "what does the game of life sound like?"

The main objective was clear, to find a non-bijective way to relate a state of the game of life with a sound, but I also decided to impose some restrictions on myself:

- As far as possible, if several acceptable options are found, choose the one that is closest to the natural intrinsic rules of music. Both nature and music have rules. Sometimes these rules are self-evident, other times they are imposed, and other times they are unwritten. But the undeniable fact is that opting for any candidate without understanding this would yield dissonant and not at all pleasant results according to our subjective perception.

- To not take sides, that is, not to decide what was before neither the sound nor the state of the game.

Various mathematical approaches arose to solve the problem. Apply group theory, my forever-beloved mathematical analysis, or algebraic-combinatorial methods. I applied Occam's razor to choose the latter, with the intention of not excessively slowing down the results obtaining. Whatever the intermediate calculation is , the basic idea is to loop through the array and return a numeric value. A module (% x) operation will be applied to this value later in music.py, where "x" represents the number of notes or options that correspond to the chosen musical approach.

The *mod* operation is necessary given the infinity of frequencies that could be modeled with the game of life. All that combinatorial explosion is limited by just contemplating a series of musical notes, harmonic progressions, or rhythmic blows. But since this operation is outside of maths.py, it will be covered later.

Considering all the previous information, strange as it may sound if the intention is to get closer to nature, it is best to rely on prime numbers and symmetry. Basic scheme for the finally chosen association function looks like this:

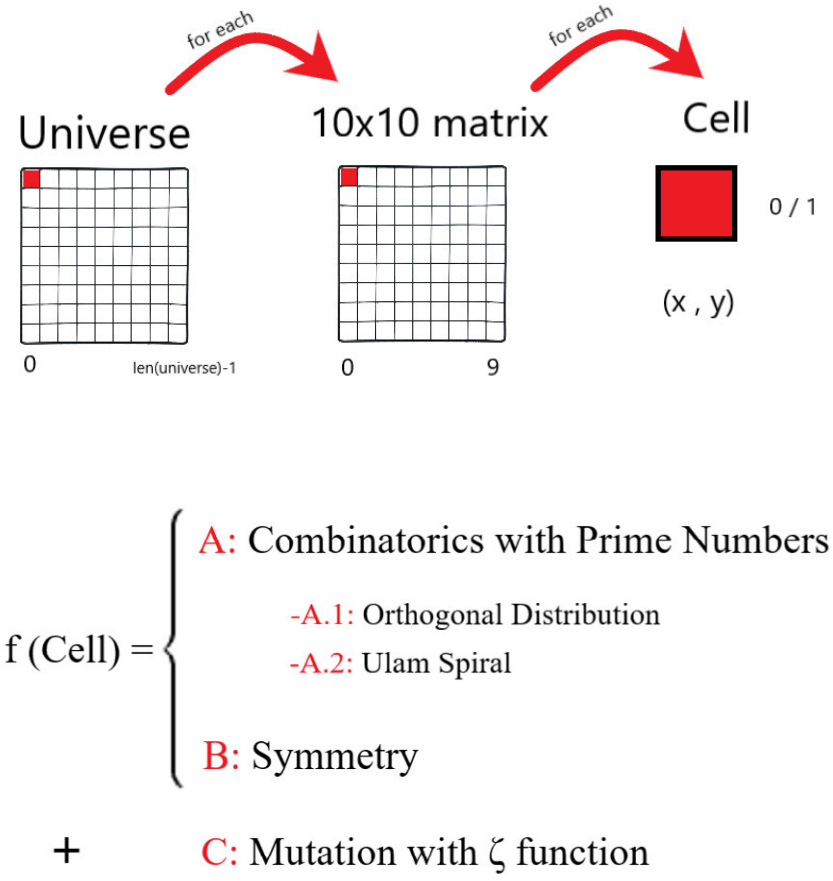



Figure 4.1.1: Basic Scheme for association function in maths.py

Applying a non-recursive divide-and-conquer technique was an obvious design decision. If the weight of the cell is affected by the prime numbers, the frequency of appearance of these is reduced as the size of the matrix increases. Therefore, setting up an association function for the entire matrix (of dimensions to be chosen by the user) was simply not very intuitive.

If ostensibly the size of the universe is a parameter that must be controlled to be divisible into n integer matrices of some dimension, it seems that dealing with multiples of 10 is convenient. On the other hand, in the first hundred integers twenty-five primes can be found, exactly a frequency of 1/4. If the prime numbers should have a higher weighting, that frequency is perfect for our purposes. Once we have one cell, as it is shown in the figure 4.1.1, the function goes through two different phases. In each one, the final value is updated with a certain weighting until the end when it is mutated once.

Except for the imports of the necessary libraries and for the first time since the project's code was introduced, due to consistency with the precise order in which the function performs its calculations, the python maths.py file will not be explained in strict order.

```
1   from sympy import *
2   import numpy as nm
3   from mpmath import *
```

Being maths.py a module dedicated to the mathematical logic responsible for the association function, the three libraries finally chosen do not surprise at all:

- **SymPy:** is a Python library for symbolic mathematics and represents one of the standard computer algebra systems (CAS) for the programming community in this language. In this case, I just used its `isPrime()` function to be able to distinguish them from composed integer numbers.

- **NumPy:** is the standard scientific computing package in Python. Its multidimensional array object was useful for my calculations, but particularly in mutation phase which will be analyzed later, its method to return the absolute value of a complex number was crucial.

- **mpMath:** is a Python library for real and complex floating-point arithmetic with arbitrary precision. It provides functions that allow you to work with various real and complex functions in a simple and intuitive way. Being Riemann's Zeta function the central topic of my first final degree project [16], I could not resist the temptation to somehow introduce it into my association function.

But following the scheme of figure 4.1.1, the first actions carried out by the association function are those of non-recursive divide and conquer. Given a universe whose size was correctly controlled to be multiples of ten, the code bellow divides it in 10x10 arrays and process them to actualize the final value:

```
131 # Divides the universe in 10x10 arrays and return the combined value
132 def divideEtImpera(universe,u):
133 
134     height = len(universe)
135     width = len(universe[0])
136 
137     array = [[ 0  for i in range(10)] for j in range(10)]
138     value= 0
139 
140     i = 0
141     j = 0
142     for m in range(int(width/10)):            # Value is updated m x n =  len(universe)/10 *2
143         for n in range(int(height/10)):
144             for y in range(10):        # Each array evaluated individually has 10x10 elements
145                 for x in range(10):
146                     array[y][x]=universe[y+j][x+i]
147 
148             value+=associationFunction(array,u)
149 
150             i+=10
151         i=0
152         j+=10
153 
154     value=mutate(value)
155     return value
```

In a universe where the weight and height dimensions coincide, a total of $\left(\frac{len(universe)}{10}\right)^2$ 10x10 arrays must be analyzed independently.

The `divideEtImpera()` function starts by declaring an auxiliary array which later will contain the array to be managed in each iteration. Next, it proceeds to create four nested loops that work like this (given a 30x30 universe as example):

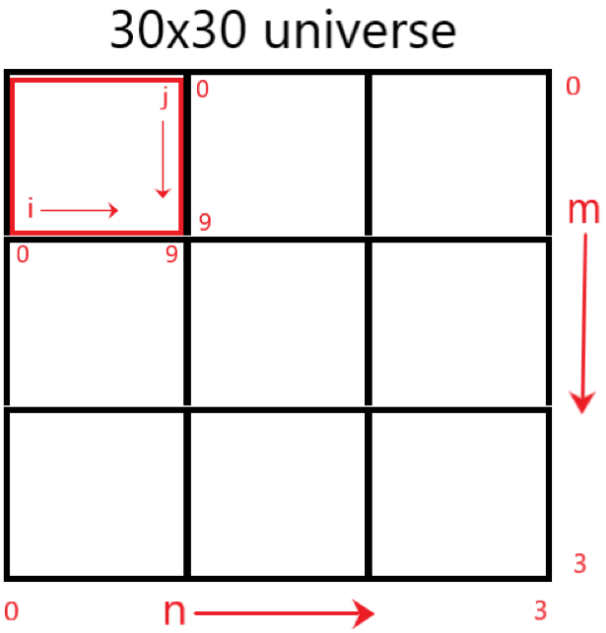



Figure 4.1.2: Iterators in the four-nested-for loops in divideEtImpera()

In order to access the corresponding arrays, it is enough to correctly indicate the position of the first element (located in the upper left corner) of each one. Therefore, sometimes the iterators “i” and “j” must be updated in intervals of ten. Once you have the auxiliary array with the hundred elements of the universe completed, the first phase begins:

### A: Combinatorics with Prime Numbers

I only knew that I wanted to use prime numbers, but now, how to place them is another story. In addition, the weighting associated with each value had to vary according to the module suggested by each musical approach. Depending on the chosen approach, the final value will distinguish between a different number of notes or different sounds. Two problems then arise, how to update the value and how to locate the prime numbers.

The first one has an obvious and crude solution, instead of updating by a fixed value, it will update in units, that is, a "*u*" unit decided by the musical approach according to the number of possibilities to model.

The second, on the other hand, placed me in a dichotomy that I resolved in a cowardly way, without opting for any option and combining both. The first distribution that I considered to distribute the primes is the orthogonal one given by the indexation of the array itself. As shown in the following figure:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
| 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 |
| 30 | 31 | 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 |
| 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47 | 48 | 49 |
| 50 | 51 | 52 | 53 | 54 | 55 | 56 | 57 | 58 | 59 |
| 60 | 61 | 62 | 63 | 64 | 65 | 66 | 67 | 68 | 69 |
| 70 | 71 | 72 | 73 | 74 | 75 | 76 | 77 | 78 | 79 |
| 80 | 81 | 82 | 83 | 84 | 85 | 86 | 87 | 88 | 89 |
| 90 | 91 | 92 | 93 | 94 | 95 | 96 | 97 | 98 | 99 |

Figure 4.1.3: Orthogonal Distribution of prime numbers in 10x10 array

However, the second distribution is much more interesting. Given the contributions of Stanislaw Ulam to the theory of cellular automata, it seemed mandatory to me to take advantage of a concept of his own invention: the Ulam Spiral, a graphical depiction of prime numbers that he found casually in 1963 while being bored at a scientific meeting. It is constructed by writing the positive integers in a square spiral and specially marking the prime numbers, which group forming mysterious diagonals instead of a random distribution.

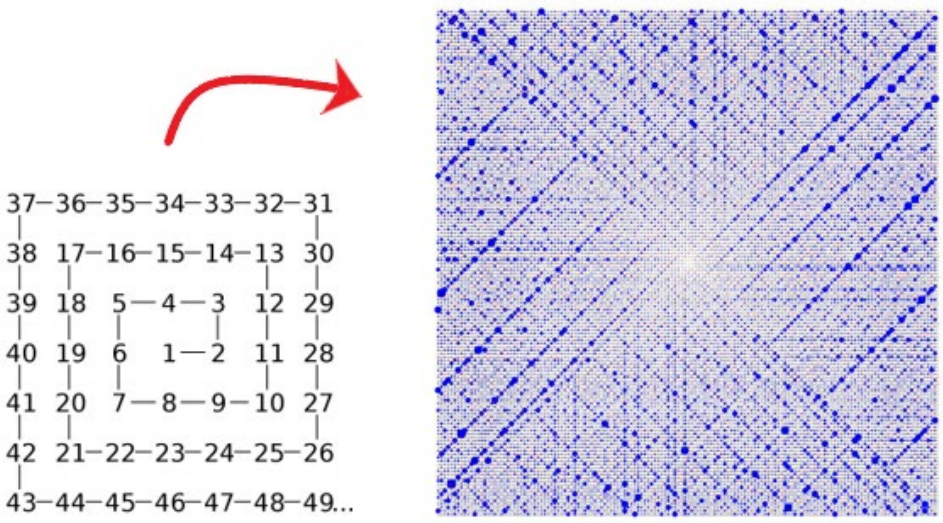



Figure 4.1.4: Construction and prime diagonals in Ulam Spiral

The spiral is directly related to many theorems and conjectures in various areas of mathematics, such as the Landau's problems, the Goldbach's conjecture, the prime number theorem, the theory of prime-generating polynomials, etc... Either way, I must force myself to remember that the project on pure mathematics was the first [16], while here I can hardly comment the brief surface of some interesting aspects of Ulam's Spiral without going into detail. Although with just the first hundred integers the outcome is not too revealing, applying the aforementioned spiral to the matrix in maths.py results in this distribution:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 100 | 99 | 98 | 97 | 96 | 95 | 94 | 93 | 92 | 91 |
| 65 | 64 | 63 | 62 | 61 | 60 | 59 | 58 | 57 | 90 |
| 66 | 37 | 36 | 35 | 34 | 33 | 32 | 31 | 56 | 89 |
| 67 | 38 | 17 | 16 | 15 | 14 | 13 | 30 | 55 | 88 |
| 68 | 39 | 18 | 5 | 4 | 3 | 12 | 29 | 54 | 87 |
| 69 | 40 | 19 | 6 | 1 | 2 | 11 | 28 | 53 | 86 |
| 70 | 41 | 20 | 7 | 8 | 9 | 10 | 27 | 52 | 85 |
| 71 | 42 | 21 | 22 | 23 | 24 | 25 | 26 | 51 | 84 |
| 72 | 43 | 44 | 45 | 46 | 47 | 48 | 49 | 50 | 83 |
| 73 | 74 | 75 | 76 | 77 | 78 | 79 | 80 | 81 | 82 |

Figure 4.1.5: Ulam Spiral-based Distribution of prime numbers in 10x10 array

The logic related to the first stage considers whether the cell in question is a prime number or not, in addition to paying special attention to the presence of any pair of twin prime numbers (according to both distributions). While exclusively in the first distribution, one unit is added for each living cell on a diagonal with prime position.


Figure 4.1.6: Logic for both distributions in phase A

One the one hand, the code section in maths.py for the logic common to both distributions is the following:

```
ulam_aux = [[100,99,98,97,96,95,94,93,92,91],
[65,64,63,62,61,60,59,58,57,90],
[66,37,36,35,34,33,32,31,56,89],
[67,38,17,16,15,14,13,30,55,88],
[68,39,18,5,4,3,12,29,54,87],
[69,40,19,6,1,2,11,28,53,86],
[70,41,20,7,8,9,10,27,52,85],
[71,42,21,22,23,24,25,26,51,84],
[72,43,44,45,46,47,48,49,50,83],
[73,74,75,76,77,78,79,80,81,82]]

ulam = nm.array(ulam_aux)
```

```
# Mathematical relationship between the state of the universe and the chosen module
def associationFunction(universe,u):
    height = len(universe)
    width = len(universe[0])

    value=0
       ### Combinatorics with prime numbers and Ulam Spiral

    i=1
    for y in range(height):
            for x in range(width):
                if(universe[y][x]==1) and (isprime(ulam[y][x])):    # Alive Prime in Ulam Spiral
                        value+=3*u
                if(universe[y][x]==1) and (not isprime(ulam[y][x])):    # Alive Cell in Ulam Spiral
                        value+=1*u
                if(universe[y][x]==1) and (isprime(i)):    # Alive Prime
                        value+=3*u
                if(universe[y][x]==1) and (not isprime(i)):    # Alive Cell
                        value+=1*u
```

Here, the typical double-nested `for` is clearly shown looping through the whole array and updating the value for each base case: first, living cells in prime or compound positions, and a little later (out of the `for` loop), the case that both twin prime numbers are alive in some distribution.

```
# Both Twin Prime Numbers Alive
if(universe[0][2]==1) and (universe[0][4]==1):
        value+=4*u
if(universe[0][4]==1) and (universe[0][6]==1):
        value+=4*u
if(universe[1][0]==1) and (universe[1][2]==1):
        value+=4*u
if(universe[1][6]==1) and (universe[1][8]==1):
        value+=4*u
if(universe[2][8]==1) and (universe[3][0]==1):
        value+=4*u
if(universe[4][0]==1) and (universe[4][2]==1):
        value+=4*u
if(universe[5][8]==1) and (universe[6][0]==1):
        value+=4*u
if(universe[7][0]==1) and (universe[7][2]==1):
        value+=4*u
```

```
# Both Twin Prime Numbers Alive in Ulam Spiral
if(universe[4][5]==1) and (universe[4][3]==1):
        value+=4*u
if(universe[4][3]==1) and (universe[6][3]==1):
        value+=4*u
if(universe[5][6]==1) and (universe[3][6]==1):
        value+=4*u
if(universe[3][2]==1) and (universe[5][2]==1):
        value+=4*u
if(universe[4][7]==1) and (universe[2][7]==1):
        value+=4*u
if(universe[6][1]==1) and (universe[8][1]==1):
        value+=4*u
if(universe[1][6]==1) and (universe[1][4]==1):
        value+=4*u
if(universe[7][0]==1) and (universe[9][0]==1):
        value+=4*u
```

The positions (x,y) of the cells involved have been previously saved in order to avoid introducing extra sentences that check the twin prime condition in each iteration of the loop, with the intention of speeding up substantially as much as possible the execution of the association function in each 10x10 array.

On the other hand, the code for the exclusive operations of the first distribution (which add one unit for each diagonally adjacent alive cell in prime position) can be found by completing the previously introduced loop:

```
        try:
            if(universe[y-1][x-1]==1) and (isprime(i-(height-1)-1)):   # Alive Prime in upper left diagonal
                value+=1*u
        except:
            value+=0
        try:
            if(universe[y-1][x+1]==1) and (isprime(i-(height-1)+1)):   # Alive Prime in upper right diagonal
                value+=1*u
        except:
            value+=0
        try:
            if(universe[y+1][x-1]==1) and (isprime(i+(height-1)-1)):   # Alive Prime in lower left diagonal
                value+=1*u
        except:
            value+=0
        try:
            if(universe[y+1][x+1]==1) and (isprime(i+(height-1)+1)):   # Alive Prime in lower right diagonal
                value+=1*u
        except:
            value+=0

        i+=1
```

where variable “i” is the pointer responsible for placing the prime cells and, therefore, must be updated at the end of each iteration. To avoid declaring the matrix with the orthogonal distribution, as was done with the Ulam spiral, the iterator “i” must be judiciously used to access the diagonal cells. First of all, the division into classes previously used for boundary cases is not essential and try/exception statements can be used instead. Secondly, the notion of adjacency used could be understood as the following relationship between range-1 Moore’s neighborhood and Neumann’s neighborhood:

$$(Range1\ Moore's\ N.) - (\ Neumann's\ N.)$$

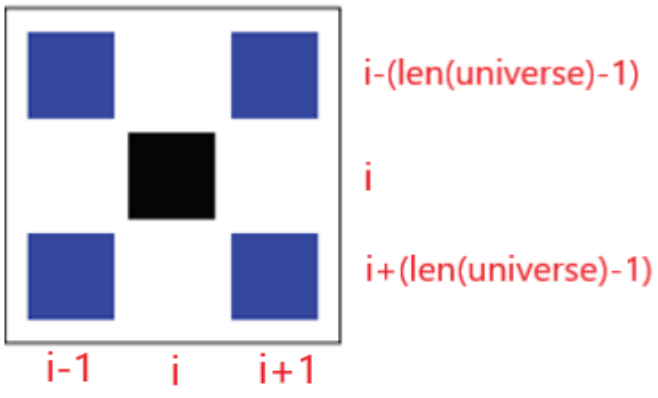



Figure 4.1.7: Notion of adjacency in Phase A

The `associationFunction()` in maths.py ends with phase B: Symmetry.

**B: Symmetry**

Symmetry is one of the ubiquitous concepts in nature and one of the reasons why Renaissance art occupies a referent place in my subjective perception of artistic beauty. If failure to find an association function sufficiently respectful of the rules of music is inevitable, I don't want to be accused of negligence for not introducing some basic axes of symmetry.

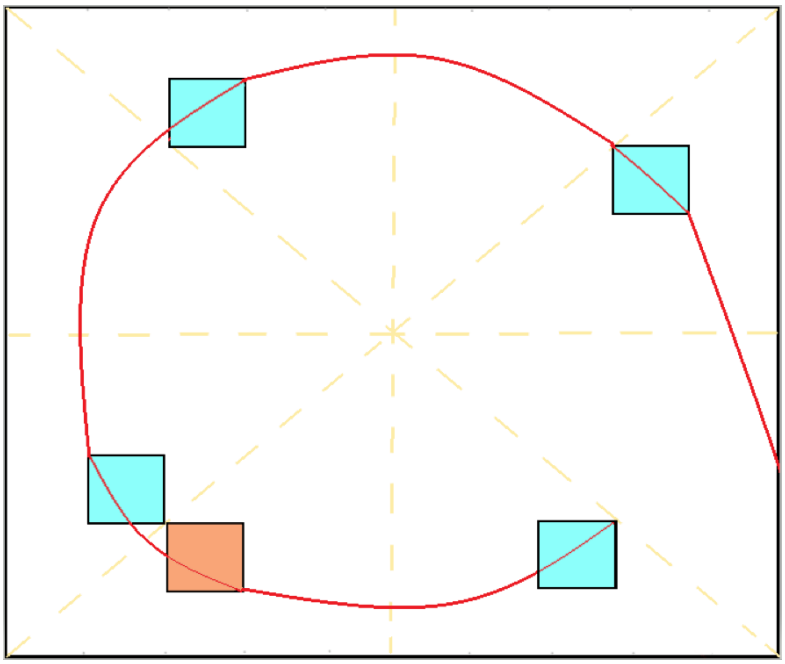


Figure 4.1.8: The four exes of symmetry in phase B

The four axes are the most basic considering that the matrix of the universe is a square: the two diagonals and the vertical/horizontal axes in the middle of the side. Therefore, each cell of the 10x10 matrix will have a symmetrical relationship with three others; if both are alive, extra units will be added to the final value of the function.

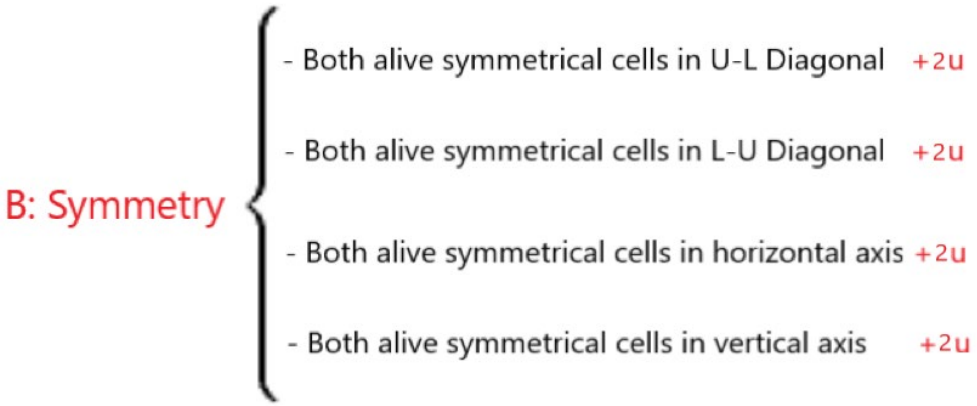



Figure 4.1.9: Logic for symmetry in phase B

The code that holds phase B and terminates `associationFunction()` corresponds with this fragment:

```
108     ### Symmetry
109
110 axis=[9,8,7,6,5,4,3,2,1,0]
111 distances = [9,7,5,3,1,-1,-3,-5,-7,-9]
112 dist=0
113
114 for y in range(len(universe)):
115         for x in range (len(universe[0])):
116             dist = axis[x]-y
117             if(universe[y][x]==1) and (universe[y+dist][x+dist]==1):       # Dextroversely, lower-upper Diagonal
118                 value+=1*u
119
120             if(universe[y][x]==1) and (universe[x][y]==1):                 # Dextroversely, upper-lower Diagonal
121                 value+=1*u
122
123             if(universe[y][x]==1) and (universe[y+distances[y]][x]==1):    # Horizontal axis
124                 value+=1*u
125
126             if(universe[y][x]==1) and (universe[y][x+distances[x]]==1):    # Vertical axis
127                 value+=1*u
128
129 return value
```

- **Lower-upper diagonal:** The coordinates (y,x) of the corresponding symmetric cell are the result of adding an integer value to the original coordinates. This value can be deduced from the vertical distance to the cell present on the diagonal. The `dist` variable in the loop calculates this distance by subtracting the “y” coordinate of the cell being evaluated from the absolute value in the `axis` array.

- **Upper-lower diagonal:** The main diagonal always results in the exchange of rows by columns, that is, the resulting symmetric cell will have the alternate coordinates of the original.

- **Horizontal/vertical axis:** The basic orthogonal symmetry also works with distances to the axis. These distances are stored in the `distances` array and affect the opposite coordinate ("y" for the horizontal axis, "x" for the vertical axis).

Line 129 is reserved for the return statement that ends with the association function.

### C: Mutation with ζ function

In most genetic algorithms, to throw off some randomness, a mutant factor can almost always be found at the end of each iteration. The mutated value, however, should not be too far from the one initially obtained by the algorithm; too much lack of control would end up diluting the association function until it is completely useless. With the idea of being able to mimic this trend in my function, I decided not to exclude mathematical analysis altogether and to use the Riemann’s Zeta function to make this final subtle change.

This mutation would follow the following formula:

$$value = value + \left\lfloor \left| \zeta\left(\frac{1}{2} + value\right) \right| \right\rfloor$$

where the result of the Riemann's zeta function in its critical line for the complex number:

$$s = \frac{1}{2} + value \cdot i$$

is calculated, to later make a round operation of its absolute value.

The last fragment of maths.py that remains to be commented is precisely the one that oversees phase C, and it is the following:

```
20  # Mutates the obtained value with Riemann Zeta Function
21  def mutate (value):
22      finalValue=value
23      if(value!=0):
24              s= nm.absolute(zeta(0.5+value*j))
25              finalValue+=round(s)
26
27      return finalValue
```

If the final value is 0, logic indicates that it should not be mutated. Since the result of the zeta function in $s = \frac{1}{2}$ is not cero, the whole mutation process is carried out just if the value is strictly positive.

To end this section, once again I found it interesting to include a heat map, the result of capturing the potential of each cell in the matrix, assuming the best conditions for the association function.

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| +7 | +6 | +10 | +12 | +7 | +12 | +7 | +10 | +8 | +6 |
| +6 | +11 | +8 | +11 | +13 | +6 | +12 | +10 | +8 | +10 |
| +8 | +8 | +9 | +8 | +7 | +6 | +8 | +8 | +9 | +12 |
| +9 | +10 | +13 | +6 | +8 | +6 | +9 | +8 | +8 | +6 |
| +7 | +10 | +8 | +16 | +7 | +10 | +7 | +10 | +8 | +6 |
| +8 | +6 | +13 | +8 | +7 | +8 | +10 | +6 | +10 | +10 |
| +7 | +14 | +9 | +10 | +8 | +6 | +6 | +8 | +8 | +6 |
| +11 | +10 | +8 | +10 | +9 | +6 | +7 | +6 | +8 | +8 |
| +7 | +10 | +8 | +8 | +7 | +8 | +7 | +6 | +8 | +10 |
| +10 | +6 | +7 | +6 | +7 | +6 | +8 | +8 | +7 | +6 |

Figure 4.1.9: Heat map for each cell's potential in maths.py

## 4.2. **music.py: from grid to sound**

At this point, the function that relates a state of the game of life to a numerical value is well defined. The problem that now arises is to relate, in turn, this obtained value with a specific sound. But again, our culture is dominated by tonal music that has evolved from the symphony orchestra to new trends and instruments. The number of possible techniques that are applied to each of the notes only pales in comparison to the huge number of instruments that are candidates for reproducing them, even more so if we accept that there is tonality in anything. For that reason, the logical first step in design phase should be limiting that combinatorial explosion.

In order to face the next approach (4.3) in a simpler way, music.py will start with 4 different instruments: piano, violin, 808-TEK bass and 808-drum kit. Following the reading order of the python module, it begins by declaring the libraries used:

```
1   import maths as mat
2   import time
3   import fluidsynth as fs
4   from mingus.containers import Note
5   import pygame
6   from pygame import mixer
```

- **Fluidsynth:** is a real-time software synthesizer based on the SoundFont2 specifications and has reached widespread distribution. It is the musical library used to produce sounds with harmonic musical instruments.

- **Mingus:** is an advanced, cross-platform music theory and notation package for Python with MIDI file and playback support. Being practically standard in Python if the program has some musical notation, its presence is not surprising. It will be important when creating the virtual keyboard and relating integer values to musical notes.

- **Pygame:** is a set of Python modules designed for writing video games, it adds functionality on top of the excellent SDL library. During my short experience developing and scoring video games, pygame was a very close and loyal friend. If the number of sounds to play is finite and relatively small, it is much more efficient to use this library instead of fluidsynth. While for harmonic instruments it's not enough, for a drum set it is perfect.

If the musical approach corresponds to 1, 4, 5 (melodic piano, melodic violin, or melodic bass respectively), the value calculated with maths.py is given by a mod 25 operation. This allows for two 12-note octaves and silence/absence of musical note.

On the other hand, if the musical approach is 2 (harmonic piano), the arithmetic will be mod 8, to obtain the seven triad chords belonging to a major/minor key plus the silence.

Finally, if the musical approach is 3 (rhythmic pattern) the operation will be mod 5 in order to obtain, as possibilities, a bass drum hit, a snare hit, the hihat and the tum (finalDrum hit) plus the silence.

Based on this idea, the virtual keyboard in music.py results in this code snippet:

```
### Initial declarations and global variables

# The keyboard can be rearranged to add complexity to the association function
notes2octavesDown = [0,Note("C", 2),Note("C#", 2),Note("D", 2),Note("D#", 2),Note("E", 2),Note("F", 2),
Note("F#", 2),Note("G", 2),Note("G#", 2),Note("A", 2),Note("A#", 2),Note("B", 2),Note("C", 3),
Note("C#", 3),Note("D", 3),Note("D#", 3),Note("E", 3),Note("F", 3),Note("F#", 3),Note("G", 3),
Note("G#", 3),Note("A", 3),Note("A#", 3),Note("B", 3)]

notes2octaves = [0,Note("C", 4),Note("C#", 4),Note("D", 4),Note("D#", 4),Note("E", 4),Note("F", 4),
Note("F#", 4),Note("G", 4),Note("G#", 4),Note("A", 4),Note("A#", 4),Note("B", 4),Note("C", 5),
Note("C#", 5),Note("D", 5),Note("D#", 5),Note("E", 5),Note("F", 5),Note("F#", 5),Note("G", 5),
Note("G#", 5),Note("A", 5),Note("A#", 5),Note("B", 5)]

notes2octavesUp = [0,Note("C", 5),Note("C#", 5),Note("D", 5),Note("D#", 5),Note("E", 5),Note("F", 5),
Note("F#", 5),Note("G", 5),Note("G#", 5),Note("A", 5),Note("A#", 5),Note("B", 5),Note("C", 6),
Note("C#", 6),Note("D", 6),Note("D#", 6),Note("E", 6),Note("F", 6),Note("F#", 6),Note("G", 6),
Note("G#", 6),Note("A", 6),Note("A#", 6),Note("B", 6)]
```

where thanks to the mingus notation, the notes and their corresponding octave can be easily deduced. On the other hand, although the chords can be overcomplicated and nuanced, for simplicity the basic triads belonging to each major and minor key have been kept. That is, given a note, the basic triad consist of a list that contains the note itself, its major/minor third and its perfect/diminished fifth depending on the key.

```
# Chords progressions for Major and Minor Tonalities
chordsMajor = [[[0, 0, 0], [Note('C',3),Note('E',3),Note('G',3)], [Note('D',3),Note('F',3),Note('A',3)], [Note('E',3),Note('G',3),Note
[[0, 0, 0], [Note('C#',3),Note('F',3),Note('G#',3)], [Note('D#',3),Note('F#',3),Note('A#',3)], [Note('F',3),Note('G#',3),Note('C',4)],
[[0, 0, 0], [Note('D',3),Note('F#',3),Note('A',3)], [Note('E',3),Note('G',3),Note('B',3)], [Note('F#',3),Note('A',3),Note('C#',4)], [N
[[0, 0, 0], [Note('Eb',3),Note('G',3),Note('Bb',3)], [Note('F',3),Note('Ab',3),Note('C',4)], [Note('G',3),Note('Bb',3),Note('D',4)], [
[[0, 0, 0], [Note('E',3),Note('G#',3),Note('B',3)], [Note('F#',3),Note('A',3),Note('C#',4)], [Note('G#',3),Note('B',3),Note('D#',4)],
[[0, 0, 0], [Note('F',3),Note('A',3),Note('C',4)], [Note('G',3),Note('Bb',3),Note('D',4)], [Note('A',3),Note('C',4),Note('E',4)], [Not
[[0, 0, 0], [Note('F#',3),Note('A#',3),Note('C#',4)], [Note('G#',3),Note('B',3),Note('D#',4)], [Note('A#',3),Note('C#',4),Note('E#',4)
[[0, 0, 0], [Note('G',3),Note('B',3),Note('D',4)], [Note('A',3),Note('C',4),Note('E',4)], [Note('B',3),Note('D',4),Note('F#',4)], [Not
[[0, 0, 0], [Note('Ab',3),Note('C',4),Note('Eb',4)], [Note('Bb',3),Note('Db',4),Note('F',4)], [Note('C',3),Note('Eb',4),Note('G',3)],
[[0, 0, 0], [Note('A',3),Note('C#',4),Note('E',4)], [Note('B',3),Note('D',4),Note('F#',4)], [Note('C#',3),Note('E',3),Note('G#',3)], [
[[0, 0, 0], [Note('Bb',3),Note('D',4),Note('F',4)], [Note('C',3),Note('Eb',4),Note('G',3)], [Note('D',3),Note('F',3),Note('A',3)], [No
[[0, 0, 0], [Note('B',3),Note('D#',4),Note('F#',4)], [Note('C#',3),Note('E',4),Note('G#',3)], [Note('D#',3),Note('F#',3),Note('A',3)],
```

```
chordsMinor = [[[0, 0, 0], [Note('C',3),Note('Eb',3),Note('G',3)], [Note('D',3),Note('F',3),Note('Ab',3)], [Note('E',3),Note('G#',3),
[[0, 0, 0], [Note('C#',3),Note('E',3),Note('G#',3)], [Note('D#',3),Note('F#',3),Note('A',3)], [Note('E#',3),Note('A',3),Note('C',4)],
[[0, 0, 0], [Note('D',3),Note('F',3),Note('A',3)], [Note('E',3),Note('G',3),Note('Bb',3)], [Note('F#',3),Note('A#',3),Note('C#',4)],
[[0, 0, 0], [Note('Eb',3),Note('Gb',3),Note('Bb',3)], [Note('F',3),Note('Ab',3),Note('B',3)], [Note('G',3),Note('B',3),Note('D',4)],
[[0, 0, 0], [Note('E',3),Note('G',3),Note('B',3)], [Note('F#',3),Note('A',3),Note('C',4)], [Note('G#',3),Note('C',4),Note('D#',4)], [
[[0, 0, 0], [Note('F',3),Note('Ab',3),Note('C',4)], [Note('G',3),Note('Bb',3),Note('Db',4)], [Note('A',3),Note('C#',4),Note('E',4)],
[[0, 0, 0], [Note('F#',3),Note('A',3),Note('C#',4)], [Note('G#',3),Note('B',3),Note('D',4)], [Note('A#',3),Note('D',4),Note('F',4)],
[[0, 0, 0], [Note('G',3),Note('Bb',3),Note('D',4)], [Note('A',3),Note('C',4),Note('Eb',4)], [Note('B',3),Note('D#',4),Note('F#',4)],
[[0, 0, 0], [Note('Ab',3),Note('B',3),Note('Eb',4)], [Note('Bb',3),Note('Db',4),Note('E',4)], [Note('C',3),Note('E',4),Note('G',3)],
[[0, 0, 0], [Note('A',3),Note('C',4),Note('E',4)], [Note('B',3),Note('D',4),Note('F',4)], [Note('C#',3),Note('F',3),Note('G#',3)], [N
[[0, 0, 0], [Note('Bb',3),Note('Db',4),Note('F',4)], [Note('C',3),Note('Eb',4),Note('Gb',3)], [Note('D',3),Note('F#',4),Note('A',3)],
[[0, 0, 0], [Note('B',3),Note('D',4),Note('F#',4)], [Note('C#',3),Note('E',4),Note('G',3)], [Note('D#',3),Note('G',3),Note('A',3)], [
```

Since it is not a chord class that is stored, but rather a triad of notes, displaying the entire array in the image is cumbersome. It is simpler in figure 4.2.1:

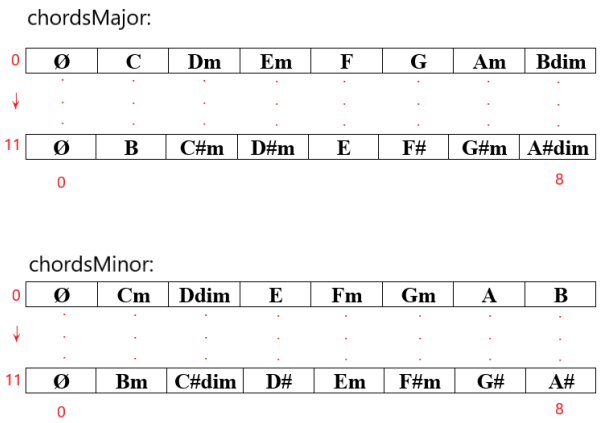



Figure 4.2.1: ChordsMajor and ChordsMinor array displayment

Ending with global variables and initial declarations, relative paths to instruments are included in .sf2 format:

```
53   # Soundfonts
54   instruments =["\yamahap45.sf2","\R-violin.sf2","\drums","\TEK bass.sf2"]
55
```

music.py continues with the declaration of the functions involved in playing sound, first with two auxiliary functions:

```
58   ### Sound Libraries and Music Generation
59
60   def getMod (mApproach):
61    if(mApproach==1):
62        return [25,4,0]                # Two octaves of twelve notes, plus silence with piano
63    if(mApproach==2):
64        return [8,2,0]                 # Assuming no modulations, each of the seven chords of a key with piano
65    if(mApproach==3):
66        return [5,1,2]                 # Bass drum, snare drum, finalD and hi hat, plus silence
67    if(mApproach==4):
68        return [25,4,1]                # Two octaves of twelve notes, plus silence with violin
69    if(mApproach==5):
70        return [25,4,3]                # Two octaves of twelve notes, plus silence with bass
71
72   def nOf(value,mApproach,mode,key):
73    if(mApproach==1):
74        print(notes2octaves[value])
75        return notes2octaves[value]
76    if(mApproach==2):
77        if(mode==0):
78                print(chordsMajor[key][value])
79                return chordsMajor[key][value]
80        if(mode==1):
81                print(chordsMinor[key][value])
82                return chordsMinor[key][value]
83    if(mApproach==4):
84        print(notes2octavesUp[value])
85        return notes2octavesUp[value]
86    if(mApproach==5):
87        print(notes2octavesDown[value])
88        return notes2octavesDown[value]
```

- **getMod:** corresponds to the basic parameter assignment. The function receives the musical approach and returns: the module applied to the value of maths.py, the value of "*u*" (unit to be added in each phase of the association function) and the instrument designated for that approach.

- **nOf:** as an abbreviation of NoteOf, corresponds to the function that returns the corresponding note from a numeric value. That is why it receives as parameters the musical approach, the mode (major or minor), the corresponding tonality and the value to be related. Then it simply examines the note at the position in the array, prints it to the screen, and returns it for later playback.

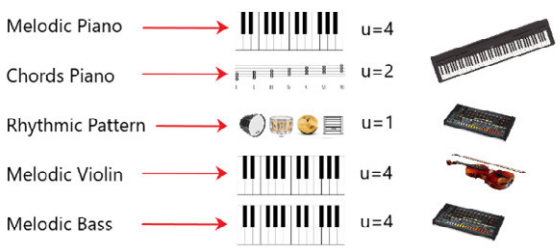



Figure 4.2.2: Parameter assignment in getMod()

And secondly with the homonymous function and protagonist of the module: `music()`. The function that appeared briefly in yawnoc.py previously but was not discussed as it was part of this section and directly responsible for playing the sound itself. In three fragments, `music()` first performs operations common to every musical approach:

```
90  def music (universe,mApproach,speed,mode,key):
91      gm= getMod(mApproach)
92      module=gm[0]
93      u=gm[1]
94      sf2 = instruments [gm[2]]
95      ins = ".\soundfonts"+ sf2
```

Then with the intervention of the pygame library in case of being in the rhythmic approach:

```
97      # Call to the hypothetic function reproduce(instrument,key(universe),speed)
98
99      if(mApproach==3):                                   #Pygame for Drum kit
100             pygame.init()
101             bassDrum = mixer.Sound(ins+"\BassDrum.WAV")
102             snareDrum = mixer.Sound(ins+"\SnareDrum.WAV")
103             hihat = mixer.Sound(ins+"\Hihat.WAV")
104             finalDrum = mixer.Sound(ins+"\FinalDrum.WAV")
105
106             n = mat.divideEtImpera(universe,u) % module
107             if(n==0):
108                 time.sleep(speed)
109                 time.sleep(speed)
110             if(n==1):
111                 bassDrum.play()
112                 time.sleep(speed)
113             if(n==2):
114                 snareDrum.play()
115                 time.sleep(speed)
116             if(n==3):
117                 hihat.play()
118                 time.sleep(speed/2)
119             if(n==4):
120                 finalDrum.play()
121                 time.sleep(speed)
```

where lines 100-104 initialize the environment and map each sound in the drum set to the corresponding .wav file and a little bit later the function `divideEtImpera()` from maths.py is then called to calculate the module of the value related to the state of the universe.

The fragment ends with the `if` sentences related to playing one sound or another from the value of "*n*". The `time.sleep()` sentences allow the ultimately control over the update speed of each state, action that was not performed in yawnoc.py in case it was necessary to distinguish between different speeds rates for each sound.

The operation scheme of pygame is so obvious that I do not think any further explanation is needed. Therefore, the `music()` function continues with the intervention of fluidsynth for the harmonic approach case:

```
122     else:
123         if(mApproach==2):
124                 s = fs.Synth()                              #Fluidsynth for chords
125                 s.start()
126 
127                 soundfont = s.sfload(ins)
128                 s.program_select(0,soundfont,0,0)
129 
130                 n = nOf(mat.divideEtImpera(universe,u) % module,mApproach,mode,key)
131                 if(n[0]==0):
132                     time.sleep(speed)
133                     time.sleep(speed)
134                 else:
135                     s.noteon(0,int(n[0]),80)
136                     s.noteon(0,int(n[1]),80)
137                     s.noteon(0,int(n[2]),80)
138                     time.sleep(speed)
139 
140                     s.noteoff(0,int(n[0]))
141                     s.noteoff(0,int(n[1]))
142                     s.noteoff(0,int(n[2]))
143                     time.sleep(speed)
144 
145                     s.delete()
```

Symmetrically to the previous fragment, lines 124-128 initialize the environment and map the corresponding instrument to its sf2 file. Even though some inopportune warnings are generated afterwards, in line 128 and due to the internal functioning of the fluidsynth library, the selection of the program to be used is required. This program contains the initialized instrument as well as the synthesizer, the channel to be used and the default midi device, particularity that should not be paid too much attention.

The current fragment continues again with the call to maths.py and ends up playing all three notes of the chord simultaneously only if there is a given chord and not the rest. The `noteon()` function literally plays the note associated with the midi number in the second parameter, while the first one is used to indicate the track and the last one for the attack speed to the note.

On the other hand, `noteoff()` disables the note played on sustain with the indicated track and midi number. The appearance of `time.sleep()` sentences can be observed again, but it is noteworthy that in the case of fluidsynth the additional statement that closes the created environment must be included (line 145).

If the module does not fit in any of the previous musical approaches, it only remains to include the last fragment of `music()`, dedicated to playing a single note with fluidsynth:

```
146     else:
147         s = fs.Synth()                              #Fluidsynth for notes
148         s.start()
149
150         soundfont = s.sfload(ins)
151         s.program_select(0,soundfont,0,0)
152
153         n = nOf(mat.divideEtImpera(universe,u) % module,mApproach,mode,key)
154         if(n==0):
155             time.sleep(speed)
156             time.sleep(speed)
157         else:
158             s.noteon(0,int(n),80)
159
160             time.sleep(speed)
161
162             s.noteoff(0,int(n))
163
164             time.sleep(speed)
165
166             s.delete()
```

This snippet is identical to the previous one except that it has only one call to the `noteon()` function since there is only one note to play.

I cannot deny that I would have loved to include more instruments or additional musical treatment. The mingus library offers the possibility of altering the timbre, the attack or the tuning of the notes, as well as being able to include more complex chords, either with sevenths, ninths or even inverted chords. On the other hand, by limiting the scales that have been included, only the Ionian and Aeolian modes are contemplated in music.py. However, including the rest of the modal scales from the Lydian mode to the Locrian mode would have introduced many more possibilities.

If I keep any kind of regret in any section of the project, it should be in the first approach. For simplicity and a pseudo-correct statistical distribution of the resulting sounds, I made very relevant decisions in both maths.py and music.py. As a result, all these possibilities discussed above could not be included in music.py to avoid unpleasant dissonances, and the chosen association function in maths.py does not have all the mathematical complexity that I would have liked, and just counts and ponders without any imagination.

It comforts me to think that from the point of view of a developer user, including new instruments or relevant updates can be very easy. This is because the code is highly understandable and most changes would have to be made in one module without needing to affect the rest. Despite everything, I think my redemption arc culminates by taking the possibilities of this system to the extreme with the second approach.

## 4.3. **Second Approach: from sound to grid**

In retrospect, I must accept that I did not know what I was getting myself into, my intention was not to take sides, but I did not expect the resulting difficulty. Thanks to music.py playing sound from the game of life was possible and relatively easy, but I forgot that frequently: "the converse is not true".

I realized that modeling a complete song is very complex, so much so that finding a single initial configuration that results in the entire song is undoubtedly impossible due to the combinatorial explosion and the statistical imperfections of the previously presented association function. For this reason, I had to make three important decisions: to model different aspects of the song in different universes of the game of life, to create a browser that looks for initial settings for a given pattern, and to accept that choosing the song was a critical phase.

Therefore, the section will be organized justifying the chosen theme, followed by a fairly exhaustive musical analysis of it and presenting the files responsible for the resulting model as a conclusion for the Python project.

### 4.3.1. ***BLACKPINK - 'Shut Down'***

There are several reasons why Shutdown ended up being chosen as the theme to model in the game of life. When I first considered what kind of song I wanted to simulate, I had a few clear requirements:

- I wanted a song that represents modern trends in music composition and production but at the same time respects and takes advantage of more classical works.

- As modeling a whole song in the same grid was a challenge, the idea was to separate harmony, melody, rhythm, and base in different Conway's Game of Life grid played simultaneously. The chosen theme must be simple in several of the scenes mentioned.

- After all, it is my project. Knowing my tendency to exclusively obsess over the things I like, I chose to use this to overcome my lack of desire to work on a new final degree project that meets my expectations but under the pressure of having less time. BLACKPINK was a must in this vast valley of possibilities.

Throughout the analysis of the song, compliance with the above requirements can be observed. The analysis will begin with the technical sheet of the song, followed by a preliminary summary of its basic characteristics and then each segment will be analyzed individually.


**Technical sheet:**

"Shutdown" is a song by South Korean girl group 블랙핑크 (RR: Beullaekpingkeu), commonly stylized in all caps or as BLΛƆKPIИK, released on September, 2022 through YG Entertainment and Interscope Records as the second single and tittle track of the group's second studio album: "Born Pink".

It is described as a hip-hop-based track with strings and an insistent trap-managed bass sound. Written by Teddy, Danny Chung and Vince finally composed by Teddy and 24, the song belongs to K-pop, Hip-hop, Rap-Pop genres and has an exact duration of 2 minutes and 56 seconds.

| "Shut Down" | |
|---|---|
| Single by Blackpink | |
| from the album *Born Pink* | |
| Language | Korean • English |
| Released | September 16, 2022 |
| Studio | The Black Label |
| Genre | Hip hop |
| Length | 2:56 |
| Label | YG · Interscope |
| Composer(s) | Teddy · 24 |
| Lyricist(s) | Teddy · Danny Chung · Vince |
| Producer(s) | 24 |

Figure 4.3.1.1: Technical sheet for Shutdown retrieved from Wikipedia.org

**Basic Characteristics:**

- The song was composed considering a tempo of 110 bpm and using as a starting point a sample of the first bars of the third movement of *"The Violin Concerto No. 2 in B minor, Op. 7"*, composed by Niccolò Paganini in 1826. That familiar sample, commonly known as *"The Campanella"* for its later appearance in the Franz Liszt's *"Grandes études de Paganini, S.141"* in 1851, is repeated insistently throughout the whole theme.

- Shutdown is developed exclusively in the key of B flat minor and there are no modulations at any point in the song.

- The simplicity in its harmony shines through its presence, since it respects the traditional and basic jump between the tonic and the dominant, with only two chords progression in almost the entire song, with B♭m as the first degree (i), F as the fifth degree (V) and the occasional appearance of G♭ as the sixth degree (VI) just in the final instrumental outro.

- The song starts from a characteristic sample with a waltz-reminiscent rhythm of 6/8 based on quarter and eight notes and ostensibly the theme will also present and maintain this rhythm. The instrumental in the background enfolds the string sample with a deep and trap-reverberate bass that is doing the same rhythmic pattern the whole time.

- However, the melody on the rap verses is more complicated. Although it follows the trail of the main late motif, it often varies depending on the specific section of the song, becoming accelerated or radically changed when there are pizzicatos in the background.

**Sampling "La Campanella":**

Niccolò Paganini was an Italian violinist and composer characterized by making the violin an extremely virtuous instrument from that point forward. Speaking now from a completely subjective point of view but clearly supported by various experts in musical analysis, in my opinion, the Italian maestro together with Frederic Chopin represent the highest references of musical romanticism.

Figure 4.3.1.2: Niccolò Paganini's Portrait

Even though one of his best-known compositions is the set of his “24 Caprices for Solo Violin Op. 1” which includes some of the most technically complex études for violin, I must accept that this project does not constitute an analysis of his whole legacy, and it is time to focus particularly on the work that sustains this section.

Here is the sheet for the beginning of *The Violin Concerto No. 2 in B minor*‘s third movement, best known as *Rondo à la clochette* (in B minor):

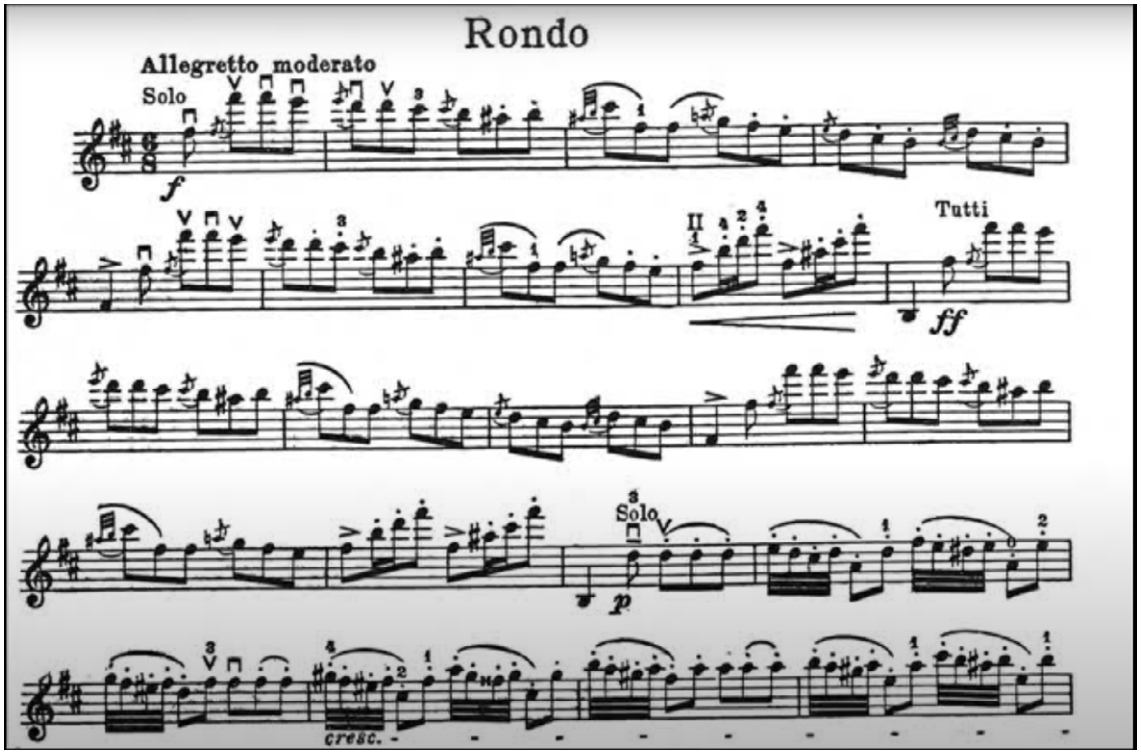



Figure 4.3.1.3: Rondo à la clochette (in B minor) fragment

Despite the possible extraction of many melodically interesting leitmotifs in this fragment, the key is to focus on the techniques common to all of them. The resource that Paganini uses obsessively and shamelessly is chromaticism: the rondo groups notes in triplets with chromatic jumps, at a distance of one or two semitones as required by the scale. The chromatic descent is continuous and hypnotizing, it seems that at all times something is about to fall apart but it is saved at the last second. Niccolo doesn't let you breathe and savor the feeling of rest that the tonic brings for too long; instead, he repeats the motif again.

The chosen tonality, infrequent, often dark and in a minor mode, together with the previously commented trend seem to have no rival. The reality, however, is somewhat different, every character in a history, whether protagonist or antagonist, must have some redeeming quality that allows us to empathize with him. The balance and contrast in this piece is achieved by the rhythm and the last arpeggio of the eighth bar.

The 6/8 gives a sensation of waltz, and the waltz do not cause fear. It intimidates in a more challenging way and compensates the tension generated by the insistent chromatic descent with a less dramatic and more innocent rhythm. On the other side, the second highlighted resource that deserves mention here is the Arpeggio. This harmonic resource consists of playing the notes that make up a chord progressively and not simultaneously. Paganini uses this technique in order to terminate the second repetition of the leitmotiv, doing the famous First-Fifth-First (i-V-i) in the eighth bar, also indicating whether the ending has been climatic or not. This resource is so charismatic that even Teddy broke with his usual rule of single sampling to introduce it in Shut Down, and it is not the only time he did it.

Apart from the commented resources, the internal structure of the piece is clear. Paganini begins by presenting the main motif of the rondo and then proceeds to explore slight variations on it, but always investigating and challenging chromatic descent. In later sections of the piece, he understands that merge means improving and not overdoing. With this idea in mind, he goes on to create a mix between the above and new inclusions such as: studying the correct way to introduce dissonances, conversations between different instruments, identifying the ideal moments to perform solos, etc. Paganini ends up perverting his piece, recovering the main motif of the rondo as the leading thread of the third movement when it is necessary in order to not lose coherence.

Focusing now at the exclusively on the main motive:


Figure 4.3.1.4: Rondo's main motif: "La Campanella"

When I first discussed Shut Down with a friend, specifically the initially controversial but later lauded decision to sample "La Campanella", I was surprised to find that he attributed the work to another composer. Convinced that I was right, we investigated the case, but I ended up understanding why many piano students were of the same opinion. Due to its complete orientation to the aforementioned instrument, this leitmotiv associated with a later étude for piano by Franz Liszt prevails in the collective imagination of pianists.

"*The Grandes études de Paganini, S. 141*" are a series of six études for piano composed by the Hungarian pianist Franz Liszt that somehow managed to transfer the technical complexity of violin in Paganini's compositions to the piano. While the original version dated from 1838, it is in the revised version of 1851 and specifically in the third étude that Paganini's original motif is recovered.

"*Étude No. 3 in G♯ minor*" (Allegretto) ("La Campanella") has a popular reputation for being one of the most difficult pieces ever written for the piano but it clearly approaches

the leitmotiv from a different angle, abandoning the chromaticism and adopting more abrupt melodic jumps, usually one or even two octaves with the right hand. In fact, the study is often used as standard practice of extremely larger jumps increasing accuracy and dexterity, in contrast to the abuse of chromaticism to which Paganini accustomed us.

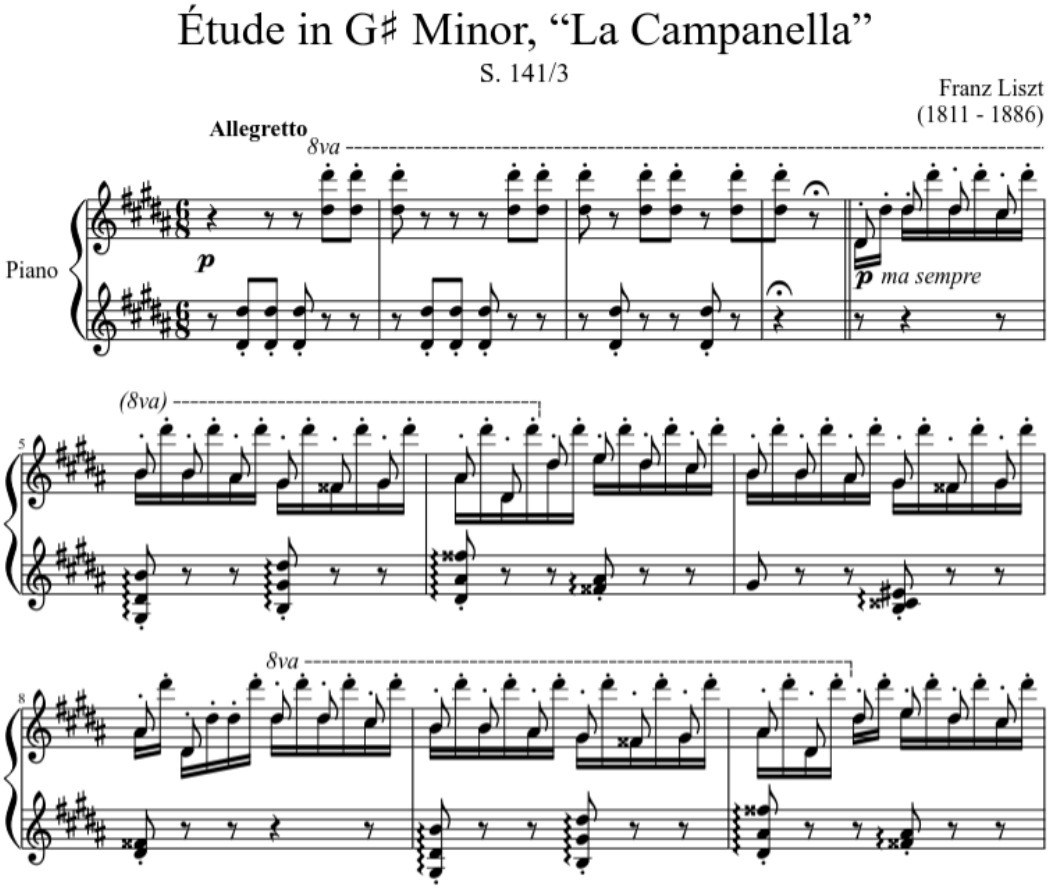



Figure 4.3.1.5: "Étude No. 3 in G♯ minor" fragment

Without going too far and as an additional comment, I wanted to add two things. First of all, it strikes me that both composers have decided to modulate Paganini's concerto in such different ways. Liszt decides to use the key of *G♯ minor* while Teddy decides to show a little bit more of respect to the original key of the piece, composing Shut Down in *B♭ minor* (the original key was *B minor*).

On the other hand, although the melody is not exactly trivial, Shut Down takes the idea of repeating the same figure over and over again to the extreme. This will make my future work on modeling the song easier, since I will be able to repeat the motif instead of following a whole melody.

This only applies to the melody in the instrumental, the verses of the song present another independent voice, something similar but with a little more intrigue.

**Melody in rap verses:**

BLACKPINK has shown that they can produce music of radically different genres. In this case, they choose to combine a series of different and dissimilar elements a priori, elements that cannot be thought of in combination, such as: one Paganini concerto and modern-rapped trap. And yet they manage to make everything sound organic, that the transition from one section to another is not forced while the members continue singing and dancing without any problem. Teddy decides to slow down the tempo and try to imagine melodies that go well with the motif, without destroying it or creating dissonances.

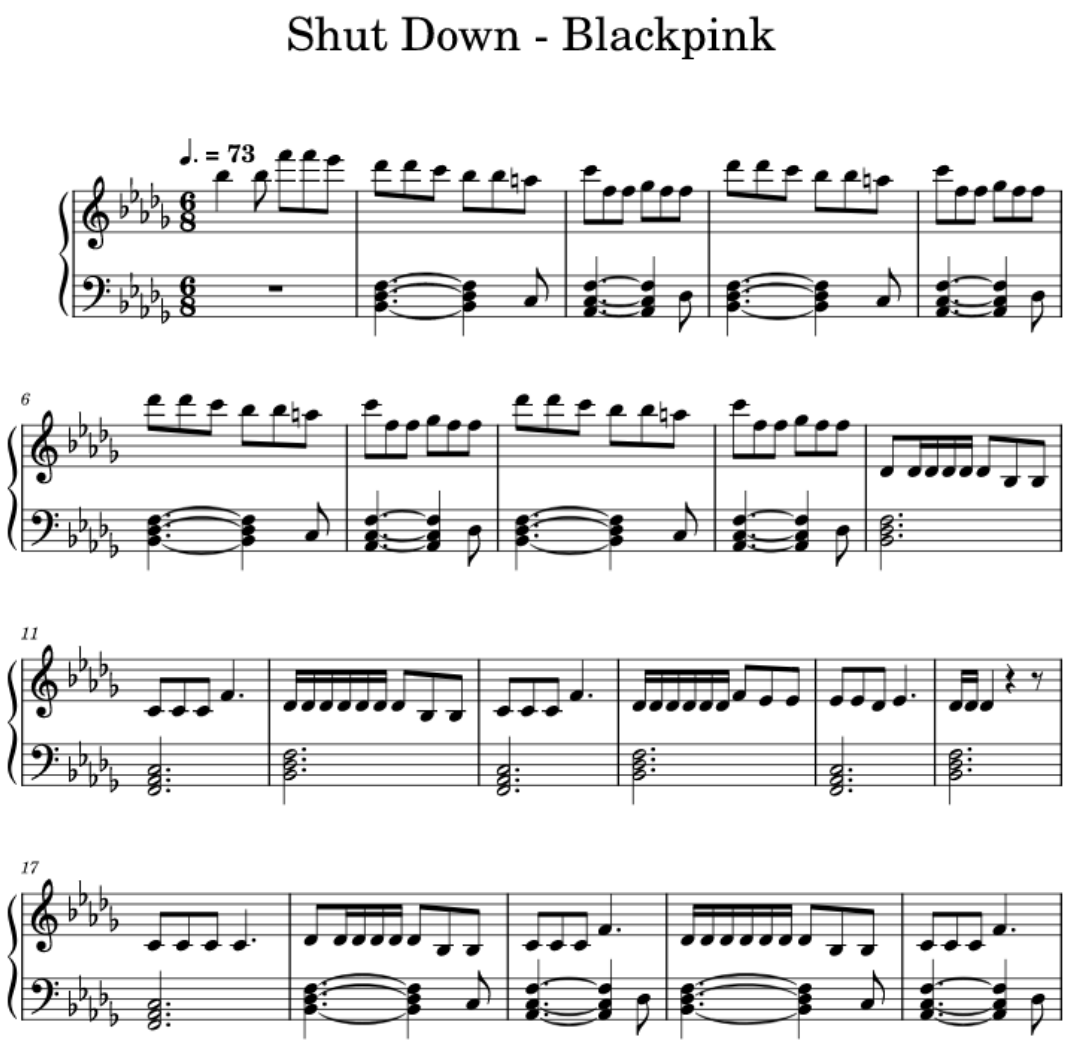



Figure 4.3.1.6: “Shut Down - Blackpink” intro and first verse

In the previous figure, the first ten bars constitute the intro of the song, and you can see the pseudo-exact copy of "La Campanella". The rest of the bars shows the melody rapped by Jennie and Lisa in the first verse of the song. The new resource that Teddy uses is to compensate the chromatic descent of the sample that is always in the background with a lifeguard, an almost pedal note that he repeats several times at the beginning of each bar.

"It may look like I'm falling, but look how I resist firmly" is the idea that he wants to represent with that melodic line. Since the lyrics of the song are dedicated to the haters and detractors of the group, in my opinion it reflects a great metaphor. Such is the strength and determination that this melody transmits that it is even allowed to play at the end of each bar with a small twist that clash with the previously built stability.

But the resources do not end here. If the reader remembers, I recently commented that Teddy had broken his usual rule of single sampling with Shut Down both a little later, in the chorus, and in the second rapped verse that serves as a pre-chorus.

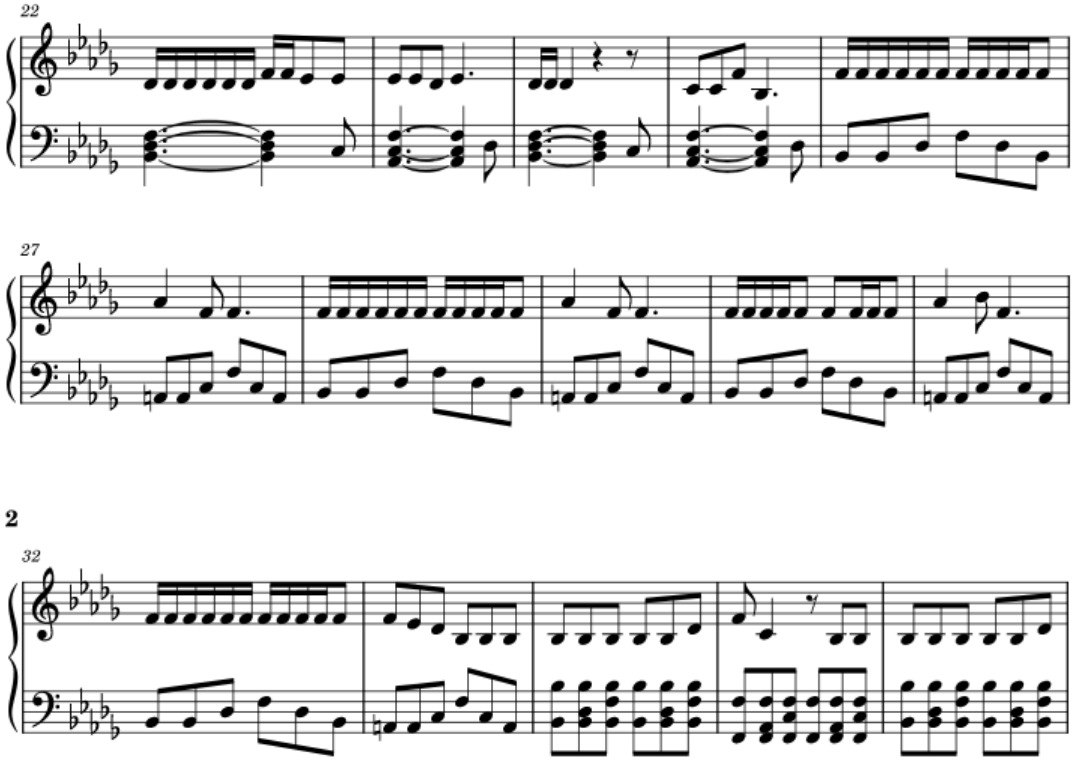



Figure 4.3.1.7: "Shut Down - Blackpink" second verse (pre-chorus)

The strategy in the melody is the same, now doubled in tempo. What is interesting about this section is the decision to create contrast. Teddy achieves this first by making the deep bass of the trap and the sample in the background disappear, to later use some subtle pizzicatos that generate a terrible tension and building-up. These pizzicatos are shown in the sheet above (figure 4.3.1.7) in the bass clef pentagram's section. Teddy takes the opportunity to create literal onomatopoeia with the aim that the music, the image and the lyrics mean the same thing becoming equivalent. He uses sounds of a garage closing or the doppler effect of an ambulance modulated to the key of the song while Jisoo and Rosé deliver their respective lines.

From bar 34 onwards the chorus begins, recovering and extolling the late motive because the song will enter its moment with more payoff.

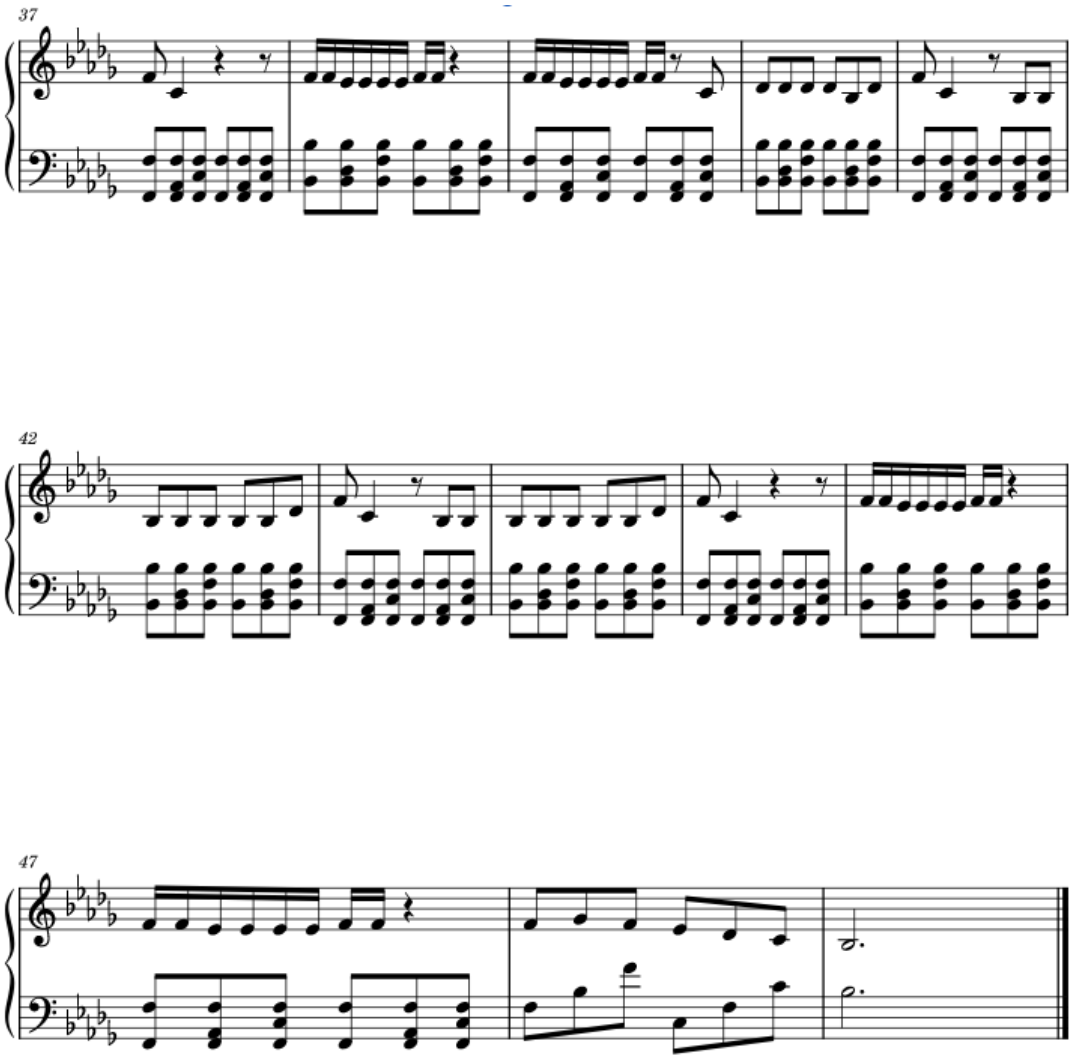



Figure 4.3.1.8: “Shut Down - Blackpink” chorus

Not even having avoided the temptation to imitate Paganini's pizzicatos, 24 and Teddy decide to also keep exactly the same arpeggio with which the rondo ended. Finalizing the chorus with the First-Fifth-First (i-V-i) that the Italian violinist introduced as the terminate signal. In the following images, thanks to Hookpad, you can observe in a more intuitive way everything previously analyzed in addition to realizing the simplicity of the harmonic progression:

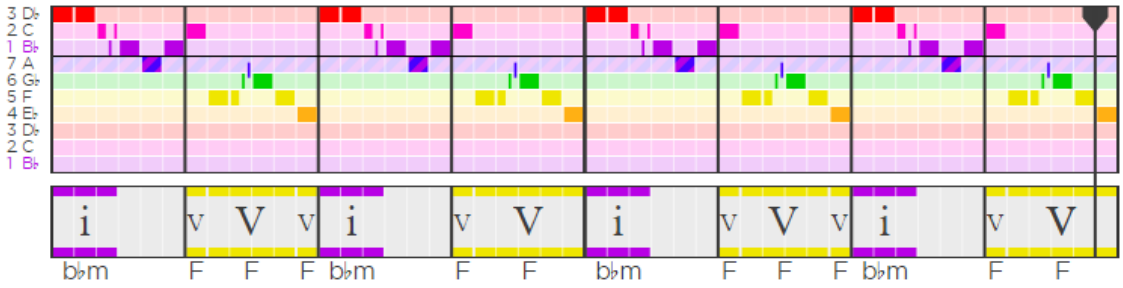



Figure 4.3.1.9: Shut Down intro in Hookpad

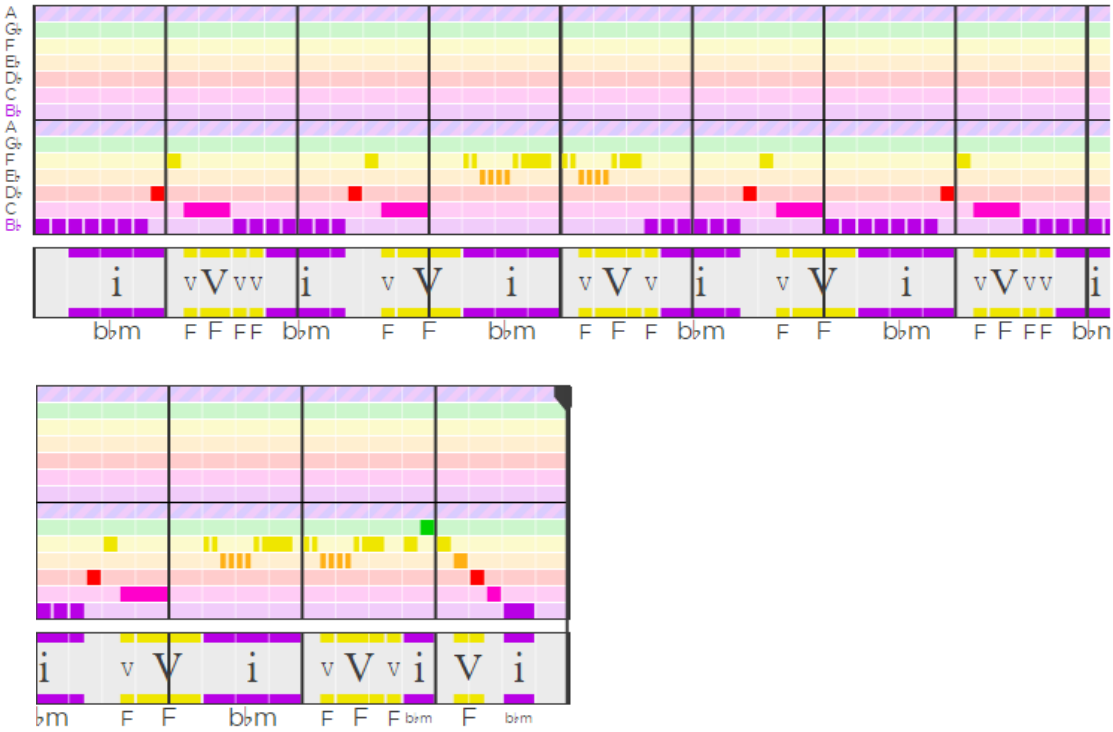


Figure 4.3.1.10: Shut Down chorus in Hookpad

The first figure (4.3.1.9) shows "La Campanella", representing the instrumental melody. However, in the second one (4.3.1.10) the melody sung during the chorus is observed.

As a curiosity, in the outro of Shut Down, the harmonic progression incorporates the sixth degree (G). The BLACKPINK girls reaffirm the newly introduced chord by chanting it while 24 designs an impassioned solo for the lead violin based on a reinterpretation of Paganini's rondo.

Having commented the most noteworthy aspects of the song's harmony and melody, to end this section it only remains to comment the spectrum, texture and rhythmic pattern that underlies the base..

**Trap base:**

Any musical genre is based on certain characteristics that make it recognizable and give it a hallmark. This set of characteristics common to all subgenres of the genre in question can be called an archetype.

In my opinion, POP is not a musical genre. Within any other well-established genre you will find pop if the characteristics of the musicalization appeal to the majority of the public. If this tendency ends up breaking any general formula or transgresses the archetype of the genre to which it supposedly belongs, that is when we will be facing a popper perversion of it.

The archetype of a genre can be well defined thanks to the delivery, habitual and essential instruments, the complexity of the harmonic progressions used or the treatment of the base in the background, among others. I will give the example of rock:

It works like a shake of country music, R&B, some Gospel and too much Blues, but always well marked by the instrument that gives it its sound: the electric guitar.

- It follows the British wave of the Beatles with bassist, singer, guitar and drummer.
- It usually uses rhythms that do not stray too far from 4/4 and 4-chord progressions with a tonal relationship.
- The delivery is usually exaggerated or shouted and rewards hoarse torn voices with very strong vocal ranges.

The trap, however, is much more difficult to analyze. This is the most mainstream portion and most intrinsically related to the urban genre within EDM (Electronic Dance Music), but it admits too many variations. This can be attributed to the fact that the trap archetype is a very specific and very small set of elements.

The delivery approaches the verses in a more narrative than melodic way, the choruses often tend to lower their intensity instead of raising it (low-energy chorus), harmonic progressions different from I-V-iv-IV are usually investigated, modulations are frequently applied, etc... Some of these characteristics may or may not be present, since minimalism is often a basic and transversal compositional principle, but where the signature of the trap is found is at the base. In order not to go around any further, I will focus on Teddy's particular strategy when it comes to creating trap-based rhythms.

Teddy Park is recognized as one of the most influential composers in South Korea. He can achieve outstanding results in radically different genres, making him an extremely chameleon-like producer. His modus operandi is to incorporate small production elements of R&B, house, reggae and trap into his work, often embellishing a simple and catchy idea. His idea of trap-instrumentals consists of the fusion of the following elements.

- A deep bass that is not always dirty produced by an 808 and with reverberation based on cutting the echo at the end of the strong pulse.

- A fanfare-like brass section in a very low key to round out the 808 hits with an added melodic twist.

- A basic percussion system made up of a constant hi hat (with slightly modified intonation), a very subtle kick and brum and a generally saturated or loud drum set that is very pleasant to listen to (sometimes just the snare drum).

Given that all these elements of the instrumental will be modeled later in the game of life, I will conclude the section with the rhythmic pattern associated with the drums, which completely conditions the rest of the background base tracks.

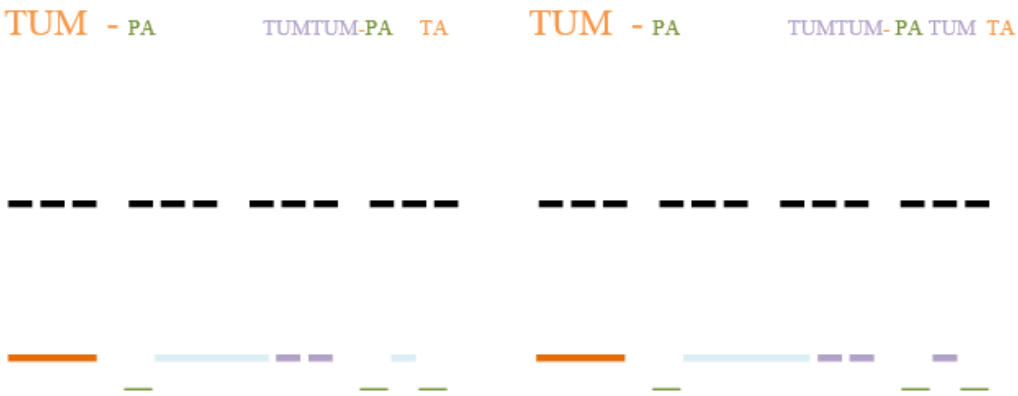



Figure 4.3.1.11: Instrumental base associated rhythmic pattern

In the figure above, the black dashes represent the 4 triplets resulting from 3/4. Just below using colors, you can see the distribution of the hits and their duration. The lower colored bars correspond to the snare hits.

The literal onomatopoeia "TUM" refers to what we might consider a traditional bass drum hit; "Pa" on the other hand, is associated with snare hits while the light blue color represents a rest of the percussion set, without any hit. The second repeat of the pattern is completely symmetrical, except that it adds an extra bass-hit/hihat on the last triplet.

In a somewhat more technical way, far from a relationship that can be extracted by simple listening, the sheet of the 808 bass and its accompaniment (working side by side with the percussion set commented previously) is this:

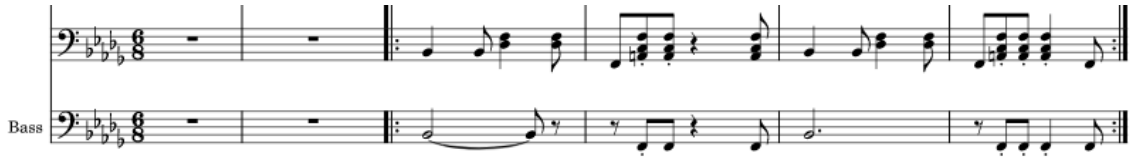



Figure 4.3.1.12: Shut Down 808 bass sheet

Teddy's objective is to reaffirm and strengthen "La Campanella". This is why he triggers the dominant on the 808 just at the exact moment in which the violin or string section precisely goes through the tonic during its chromaticism. With the same idea, he decides to avoid the brass section until the outro solo, to give priority to the leitmotif.

Again, using Hookpad for greater understanding, the instrumental base is added to the Shut Down intro.

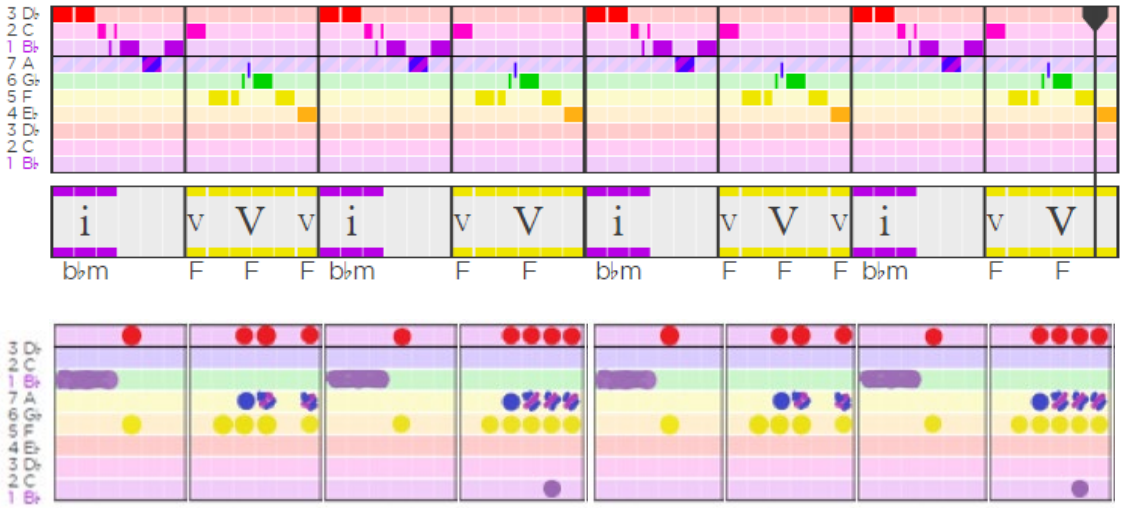



Figure 4.3.1.13: Shut Down intro (including 808) in Hookpad

### 4.3.2. *Separating mp3 song in tracks with Demucs*

Transformers and their multiple applications have been admired in the sector for a short time until now and it does not seem that interest is going to decline in the near future, quite the opposite. Within artificial intelligence and specifically in the study of neural networks and machine learning, it may be difficult to find a more significant advance than the transformer.

The first time I learned of its existence was in the brainstorming of this project. An idea flashed forth and climbed to the top suddenly, staying in place until the game of life defeated it at the last moment. This idea consisted of analyzing the mathematics involved in one of the first phases of a transformer applied to machine translation. Although the use of recurrent neural networks represented a feasible solution to this problem, there were still many problems such as memory leaks in sentences of a certain length.

Since the publication of the article: "Attention is all you need" [17] in 2017, great responsibility was given to the attention mechanisms present before only as more archaic network optimizers. When processing all the words in parallel, the transformers had to save the positional information of the words in order not to lose that key factor, especially in languages where words at the beginning or end of the sentence change the whole meaning. My intention was to propose functions with hypothetically slightly-better statistical results than those used in the initial wave phase.

Already working on the current project, I unexpectedly ran into the transformers again when I was looking to separate Shut Down into tracks. Choosing this repository to carry out this function was one of the least considered decisions of all the work, since it was a program that used artificial intelligence made in Python and that allowed me to redeem myself for abandoning the transformers. Verbatim, as the developers explain:

Demucs is a state-of-the-art music source separation model, currently capable of separating drums, bass, and vocals from the rest of the accompaniment. Demucs is based on a U-Net convolutional architecture inspired by Wave-U-Net. The v4 version features Hybrid Transformer Demucs, a hybrid spectrogram/waveform separation model using Transformers. It is based on Hybrid Demucs (also provided in this repo) with the innermost layers are replaced by a cross-domain Transformer Encoder. This Transformer uses self-attention within each domain, and cross-attention across domains. The model achieves a SDR of 9.00 dB on the MUSDB HQ test set. Moreover, when using sparse attention kernels to extend its receptive field and per source fine-tuning, we achieve state-of-the-art 9.20 dB of SDR.

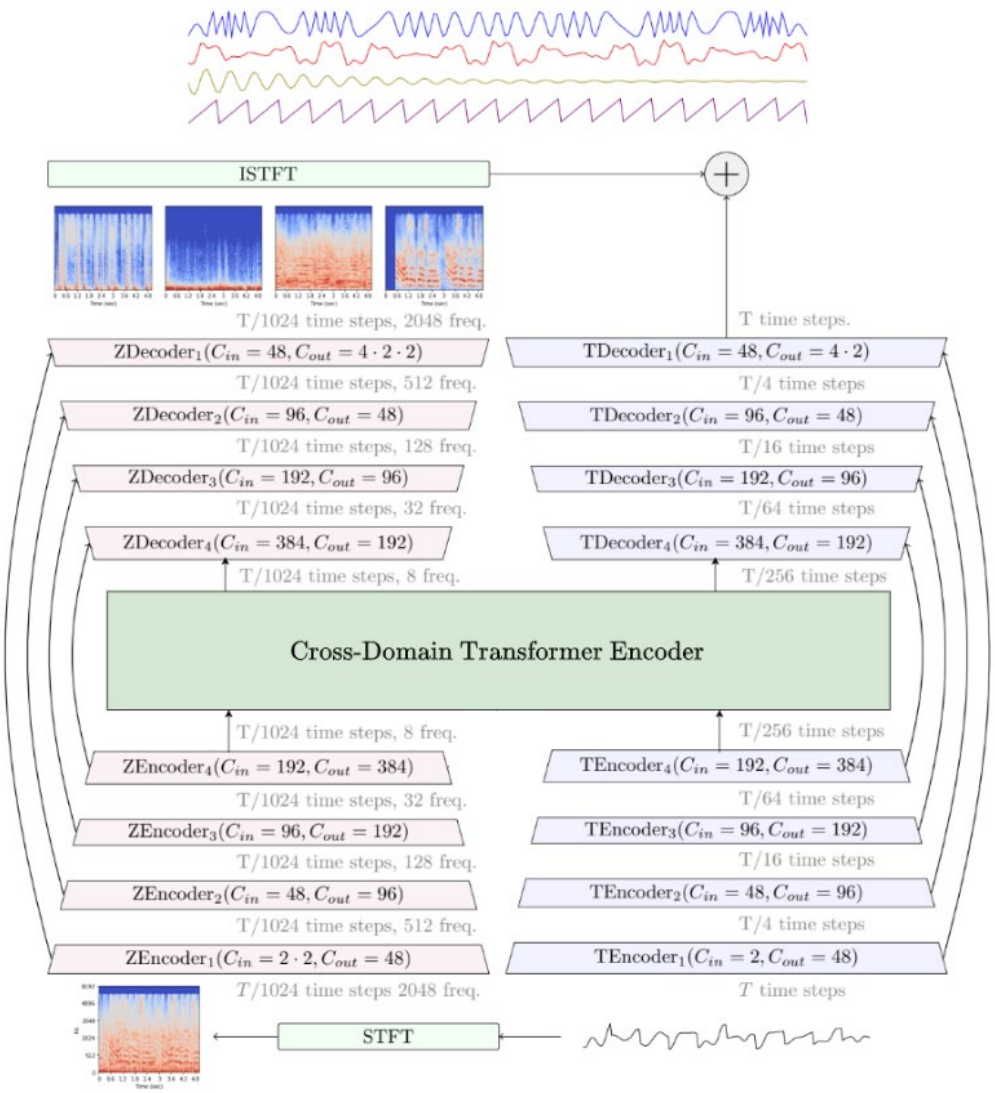



Figure 4.3.2.1: Demucs operation diagram retrieved from its repository

**Using Demucs:**

Once installed the dependencies and the necessary requirements for Demucs to work, I directly tested the transformer in action. The result was so satisfactory that I definitely have to admit some frustration at not contemplating this in my initial planning for the project.

Shut Down's musical analysis was carried out prior to the particular use of this tool, with methods much more related to brute force, removing each section by ear to later verify it on the keyboard. Like the majority of the population, I don't have perfect pitch or extensive musical training since I was a child. The little knowledge about music theory that I have, I learned it in a self-taught way and by vocation. There are several voracious methods to extract the melody and harmony of a piece using some references, but, of course, the time that the application of these techniques took me was somewhat longer than if I already had the separate tracks.

In retrospect, I don't even regret it. Those moments were by far the most satisfying stage of the project and worth every second. The mp3 file used came directly from BLACKPINK's official music video, retrieved from this link on YouTube:

https://www.youtube.com/watch?v=POe9SOEKotk

And the execution of Demucs resulted in 4 different tracks as shown in the figure below:

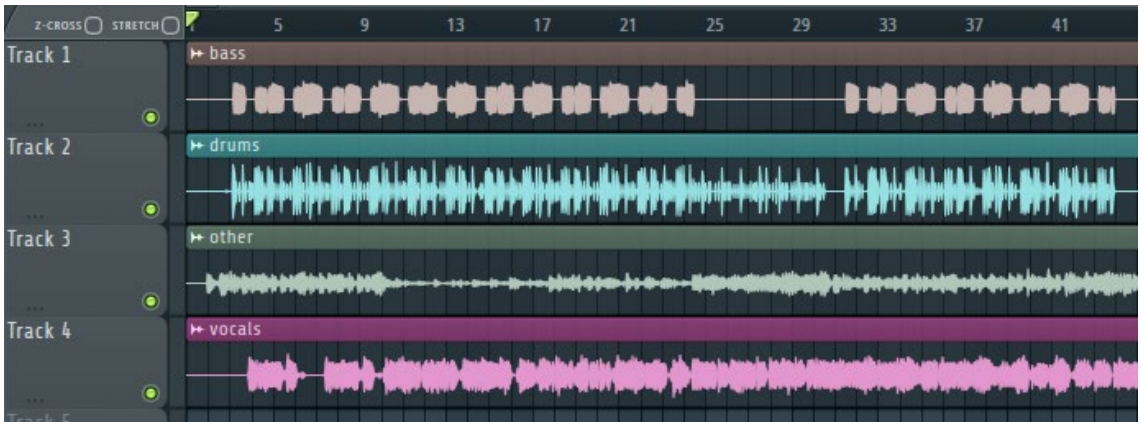



Figure 4.3.2.2: Shut Down separated tracks (intro and Chorus) in FLStudio 20

**bass:**

It contains the hits of the deep bass 808 used in the instrumental base. In the figure above you can see some sections where it disappears and they correspond to the pizzicato-based pre-choruses and the ending of each chorus, as previously discussed.

**drums:**

It contains all the instrumental accompaniment of the 808, made up of the drums (particularly the snare drum hits, bass drum hits, kicks and brums), the hihat and the additional sounds and onomatopoeia. It is on this track where the trap rhythmic pattern of the song can be best observed.

**other:**

It contains all the elements that artificial intelligence has failed to categorize as bass, drums or vocals. In this case, it brings together the violin and the string section dedicated to the leitmotif, as well as the fanfare-like brass section of the outro.

**vocals:**

Again, as the name suggests, it contains the vocals from the song. The rapped verses, the more melodic chorus and the choirs of the intro and outro are included here.

In addition, to serving as an excuse to briefly comment on the transformers in the project, this section has constituted the last step to face the modeling of the song and the second approach. Despite having separate sheets and tracks, it is necessary to indicate that this is far from being enough due to certain problems:

- In the melody there is no homogeneity in the duration of the notes. This requires deciding between two initial approaches. On the one hand, divide the track to be modeled into several subtracks that contain notes of the same length and play them in parallel. On the other hand, look for patterns in the game of life that persist in the same class when one note lasts longer than others. Despite this, especially with the violin, there are various techniques that cannot be easily imitated, so the result will not be entirely faithful to the original.

- The number of different instruments and textures used in the song is quite considerable, either you assume a generic set of instruments and model with them, or you try to sample one note from the original song to chop the others, getting more timbre accuracy. For vocals, this problem is magnified, making it impossible to imitate each artist's particular timbre, vibrato, and register. The final decision will be to use the piano to model the melody of the song, or to play the vocals track synchronously.

- Perfection is unattainable, it is right in front of us constantly but it always turns out to be a utopia. The truth is filtered through our own experiences and that is universal truth. I won't be modeling Shut Down, but my own projection of it.

### 4.3.3. *finder.py , keeper.py*

At the very beginning, the idea was to look for initial setups (preferably Methuselahs) that could exactly mimic the sheet music for each Shut Down track. I admit some ingenuity on my part, as testing each methuselah by hand and simple brute force would have been a waste of time. The alternative, however, was much better: create a browser that would do the work for me and try to make the module not biased towards my current needs. In other words, make it equally valid to find other melodies at another time.

When it comes to looking for configurations, graphing each state in the game of life is not necessary at all, much less playing sound. That is why you would only need access to the association function present in maths.py and the actualize.py methods. Clearly inspired by the famous expression: "finders keepers", I decided to split the browser into two different python modules:

- **finder.py:** is the module that the user must invoke to start the search. Therefore, it must be responsible for gathering all the parameters to be considered with input/output statements. Once it has everything it needs, it calls in its partner to do the hard work.

- **keeper.py:** is the module that contains the search algorithm itself. Responsible for both the calls to maths.py and for searching for a specific configuration and then checking each successful search.

Given the logical order of execution, finder.py will be presented first.

**finder.py:**

```
1   import keeper as k
2
3   ###Initial declarations and global parameters
4
5   mApproach = 1
6   universeD = 100
7   searchL = 1000
8
9   values = []
10
11  allGood = [False,False,False,False]
```

After the obviously necessary import of its companion, finder.py declares some global variables. The first four are initialized so as not to generate problems, but they will be correctly assigned later according to what the user indicates.

On the other hand, `allGood[]` does deserve an additional comment. It is an array of four boolean values that will manage the correct introduction of the search parameters by the user. If one is not set to `True`, the module does not continue to the next section.

The first section is dedicated to saving the musical approach that the user indicates in the previous global variable:

```
15  ### Setting the searching Parameters
16
17  # mApproach
18  while(not allGood[0]):
19      try:
20          mApproach = int(input("\nChoose one musical approach \n\n"
21          + "\n\t1: Melodic Piano"
22          + "\n\t2: Armonic Piano"
23          + "\n\t3: Rhytmic & Drums"
24          + "\n\t4: Melodic Violin"
25          + "\n\t5: Melodic Bass\n\n"))
26      except ValueError:
27          print("mApproach must be an integer")
28          exit(-1)
29
30      if (mApproach<1) or (mApproach>5):
31          print("\nERROR: wrong musical approach, \n\nmust use a"
32          +"\n\tnumber between 1-5 for:"
33          +"\n\t[musical approach]")
34      else:
35          allGood[0] = True
```

We are faced with a basic structure for entering parameters managed by a `while` loop. Initially, the message is shown on the screen, if the character entered is not an integer, the program stops. If, on the contrary, the error is not so flagrant and you have only entered an integer outside the established range, a message is displayed as a warning and the loop returns. Only if the entered value is correct, `True` is applied to the corresponding boolean in `allGood[]` and proceeds to the next section. This structure will be repeated in each section, where only the message that is printed on the screen and the range of values allowed for each parameter will be changed.

```
37  # universe dimension
38  while(not allGood[1]):
39      try:
40          universeD = int(input("\nChoose one universe dimension \n\n"
41          + "\n\tmust be a multiple of 10\n\n"))
42      except ValueError:
43          print("universe dimension must be an integer")
44          exit(-1)
45
46      if ((universeD%10)!=0):
47          print("\nERROR: wrong universe dimension, \n\nmust use a"
48          +"\n\tmultiple of 10 for:"
49          +"\n\t[universe's Dimension]")
50      else:
51          allGood[1] = True
```

The second section stores the size for the universe that the user has set for his search. This time the correct range of values only includes multiples of 10 for the same reason this restriction was introduced in yawnoc.py.

For the third search parameter, the valid range includes multiples of 100. This can be easily figured out later, but due to the presence of a progress bar in keeper.py, the number of attempts must be a multiple of 100 for its correct update.

```
53  # search limit
54  while(not allGood[2]):
55      try:
56          searchL = int(input("\nChoose one limit for analyzed configurations \n\n"
57          + "\n\tmust be a multiple of 100\n\n"))
58      except ValueError:
59          print("search limit must be an integer")
60          exit(-1)
61
62      if ((searchL%100)!=0):
63          print("\nERROR: wrong search limit, \n\nmust use a"
64          +"\n\tmultiple of 100 for:"
65          +"\n\t[search limit]")
66      else:
67          allGood[2] = True
```

The fourth parameter is no longer so simple, since it is about the specific pattern (melodic, harmonic or rhythmic) that the user wants to find. As there are five different approaches, each one will have its particular case. Beginning with the melodic piano:

```
70  ### Setting the set of values to be achieved
71
72  v = -5
73  # 2 octaves piano
74  if(mApproach==1):
75      print("\nNotes distribution and its associated number \n\n"
76      + "\n\t■  C4  C#4 D4  D#4  E4  F4  F#4 G4  G#4 A4  A#4 B4"
77      + "\n\t0   1  2   3    4   5   6    7  8   9   10  11  12"
78      + "\n\t"
79      + "\n\t   C5  C#5 D5  D#5  E5  F5  F#5 G5  G#5 A5  A#5 B5"
80      + "\n\t   13  14  15  16   17  18  19  20  21  22  23  24"
81      + "\n\t")
82      print("■ stands for silence. Type -1 to terminate")
83
84      while(not allGood[3]):
85          try:
86              v = int(input("Introduce next note: \n\n"))
87          except ValueError:
88              print("introduced note must be an integer")
89
90          if(v<-1) or (v>24):
91              print("introduced note must be between -1 - 24")
92          else:
93              if(v==-1):
94                  allGood[3] = True
95              else:
96                  values.append(v)
```

The section here is somewhat different than in the previous cases since you have to control whether the user wants to enter a new note or simply has already finished the entire pattern. Therefore, the virtual keyboard is shown on the screen, clearly indicating which integer is associated with each note.

Since there is more room for error here, entering a non-integer character does not radically end the execution of the program, still returning to the loop as if an invalid integer had been entered. The "0" is reserved for the silence, while the "-1" to indicate the end of the pattern and the rest of the integers up to "24" to represent each note. Each valid integer is added to the array of values until the user decides to end the loop. It should be noted that the larger the pattern, the more difficult it will be to find a configuration that produces it. It is advisable to divide the melody into bars and look for a pattern for each one.

```
 98  # 7 chords
 99  if(mApproach==2):
100      print("\nChords distribution and its associated number \n\n"
101      + "\n\t            Major Tonality          "
102      + "\n\t■   I   ii  iii   IV   V   vi  viidim "
103      + "\n\t0   1   2    3    4    5   6     7    "
104      + "\n\t"
105      + "\n\t            Minor Tonality          "
106      + "\n\t■   i  iidim   III   iv   v   VI  VII "
107      + "\n\t0   1    2      3    4    5   6    7  "
108      + "\n\t")
109      print("■ stands for silence. Type -1 to terminate")
110
111      while(not allGood[3]):
112          try:
113              v = int(input("Introduce next chord: \n\n"))
114          except ValueError:
115              print("introduced chord must be an integer")
116
117          if(v<-1) or (v>7):
118              print("introduced chord must be between -1 - 7")
119          else:
120              if(v==-1):
121                  allGood[3] = True
122              else:
123                  values.append(v)
```

Except for the message that shows the integer associated with each chord, there are not many differences between this snippet and the previous one. This scenario will be replicated until the end of the module.

The distribution of the chords has been carried out respecting the fundamental triads. The roman coding respects whether the chord in question is major, minor, or diminished within the key, which are confusingly also distinguished by major and minor keys. Another of the reasons why musical nomenclature is so complicated at first for new interested parties.

It should be forbidden to be able to improvise in a G Doric scale over a G minor chord in a piece composed in D minor making it logical that people get frustrated in their first skirmishes with music theory. But I do not want to fail by not being able to contain my impulses, therefore I will return to the subject at hand.

Except for the change of octaves, and their correct layout on the screen, the cases for the violin and the bass respectively do not differ at all from the piano, there is only a slight change, nothing substantial if the pattern to be searched for is rhythmic (because there will only be 5 valid values in addition to "-1").

```
125   # Drums
126   if(mApproach==3):
127       print("\nDrum hits distribution and its associated number \n\n"
128       + "\n\t■    BassDrum   SnareDrum   HiHat   Tum(FinalDrum)"
129       + "\n\t0       1           2         3           4      "
130       + "\n\t")
131       print("■ stands for silence. Type -1 to terminate")
132
133       while(not allGood[3]):
134           try:
135               v = int(input("Introduce next hit: \n\n"))
136           except ValueError:
137               print("introduced hit must be an integer")
138
139           if(v<-1) or (v>4):
140               print("introduced hit must be between -1 - 4")
141           else:
142               if(v==-1):
143                   allGood[3] = True
144               else:
145                   values.append(v)
```

The module finder.py ends with the calls to its partner keeper.py, first the one for the search and then the call to check the result obtained, informing the user of the final result of the process.

```
199   # Calls to keeper.py
200   resul = k.findConfiguration(mApproach,universeD,searchL,values)
201
202   if (resul[0]==1):
203       print("Unsuccessful Search")
204   else:
205       print("Successful Search, Checking: \n\n")
206       k.checkConfiguration(mApproach,values,resul[1])
```

```
python finder.py
```

Figure 4.3.3.1: Running example of finder.py in Anaconda Prompt

**keeper.py:**

```
import maths as mat
import actualize as a
from music import getMod
import random
import numpy as nm
from progress.bar import Bar
```

Liars are caught very quickly, although it is not necessary to import the entire music.py module, taking advantage of the `getMod()` function is mandatory for this browser. Apart from that and from maths.py/actualize.py as previously mentioned, the only surprise is the presence of a new library: progress. But given the fact that it is a mere aesthetic import, I do not think I should go too far either, its presence is just as relevant as the presence of the progress bar on the screen, an extra gadget. The search algorithm, on the contrary, does actually need several captures:

```
###Searching

# Searching for an initial configuration that satisfies the values given
def findConfiguration(mApproach,universeD,searchL,values):
 dimXgrid = universeD
 dimYgrid = universeD
 searchLimit = searchL

 gm = getMod(mApproach)
 module = gm[0]
 u = gm[1]

 fileName = ''
 if(mApproach==1):
        fileName = 'melodic piano '                # Two octaves of twelve notes, plus silence with piano
 if(mApproach==2):
        fileName = 'armonic piano '                # Assuming no modulations, each of the seven chords of a key with piano
 if(mApproach==3):
        fileName = 'drums '                        # Bass drum, snare drum, cymbal and hi hat, plus silence
 if(mApproach==4):
        fileName = 'melodic violin '               # Two octaves of twelve notes, plus silence with violin
 if(mApproach==5):
        fileName = 'melodic bass '                 # Two octaves of twelve notes, plus silence with bass

 file = ''
```

The function starts by declaring some variables, some related to the function parameters and others like `fileName` dedicated to formatting the possible local file that would save the found configuration. The call to music.py.getMod() only serves to extract from the musical approach both the module and the unit of weight, since maths.py will be called with that information later. Other relevant variables before the search loop are:

```
configuration = [[ 0  for i in range(dimXgrid)] for j in range(dimYgrid)]
obtained = [ 0  for i in range(len(values))]

resul = nm.array(configuration)
found = False

limit = searchLimit
limitbar =limit/100
```

- **configuration:** an array that stores each universe that is being analyzed.

- **obtained:** an array with the same extension as the desired pattern, which will store the resulting pattern of each universe to compare it with the desired one.

- **resul:** variable that saves the initial state in case it generates the desired pattern, to return it as the result of the search.

- **limit/limitBar:** variables that store the number of attempts to make and its division by 100 to generate the update steps of the progress bar respectively.

Now everything is ready to execute the search algorithm, which is collected inside the progress bar using the `with` sentence and can be better understood thanks to figure 4.3.3.2:

```
with Bar('Searching', fill='', suffix='%(percent).1f%% - %(eta)ds') as bar:
 while(not found) and (limit>0):
        configuration = [[ random.choice([0,0,0,0,1,1]) for i in range(dimXgrid)] for j in range(dimYgrid)]
        resul = configuration
        ep = True
        cont = 0
        contBar = 100

        while(cont<len(values)) and (ep):
            obtained[cont]= (mat.divideEtImpera(configuration,u)) % module

            if(obtained[cont]!=values[cont]):
                ep = False

            cont+=1
            configuration = a.nextState(configuration)

        if(obtained==values):
            found=True
            nm.savetxt(fileName + str(limit)+'.txt',resul,fmt='%d')
            file = fileName + str(limit)+'.txt'

            for x in range(contBar):
                bar.next()


        if(found== False):
            if(limit % limitbar==0):
             bar.next()
             contBar -= 1

        limit -=1
```

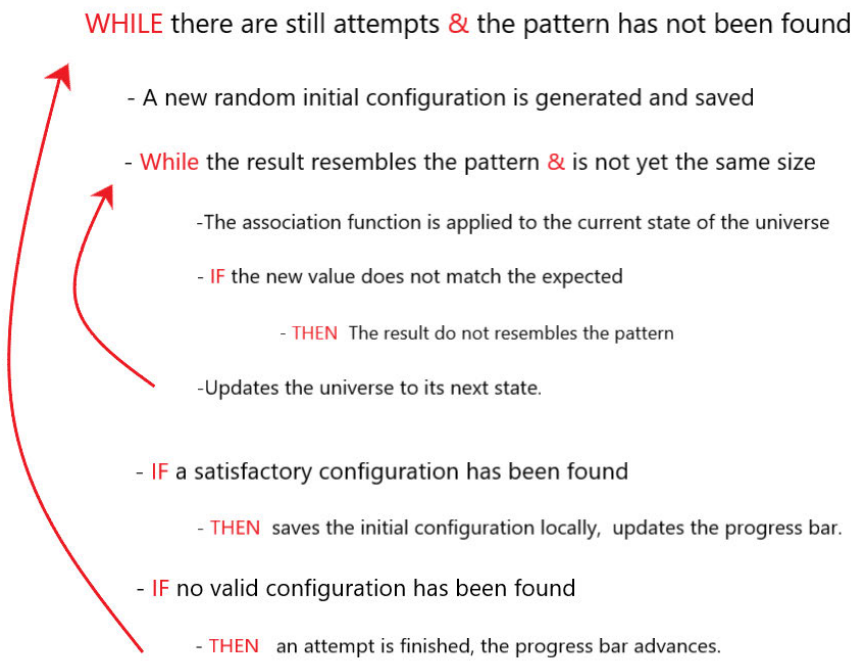



Figure 4.3.3.2: Search algorithm in keeper.py scheme

Whether or not a result is found, the keeper.py module ends up returning the name of the local file and later on declaring the function that checks if a saved configuration generates one desired pattern. Function that will be called just for successful searches.

```
78      if(found==True):
79          return[0,file]
80      else:
81          return[1,file]
82
83
84
85  ### Checking
86
87  # Checking given values
88  def checkConfiguration(mApproach,values,file):
89      gm = getMod(mApproach)
90      module = gm[0]
91      u = gm[1]
92
93      data = nm.loadtxt(file, dtype=int)
94      obtained = [ module+2  for i in range(len(values))]
95
96      for x in range(len(values)):
97          obtained[x] = (mat.divideEtImpera(data,u)) % module
98          data = a.nextState(data)
99
100     print(values)
101     print(obtained)
```

### 4.3.4. *shutdown.py and results of the model*

The inherent difficulty with chromatism resides in the fact that from one state to another in the game of life, the value of the function must differ by a fixed amount module. It has not been statistically frequent to quickly find an initial configuration for each bar. On the other hand, precisely with the intention of increasing the probabilities, the vast majority of 6/8 bars have been decided to be interpreted as two 3/4 bars. This means that from an initial configuration only matches three notes in a row before switching to the next saved configuration.

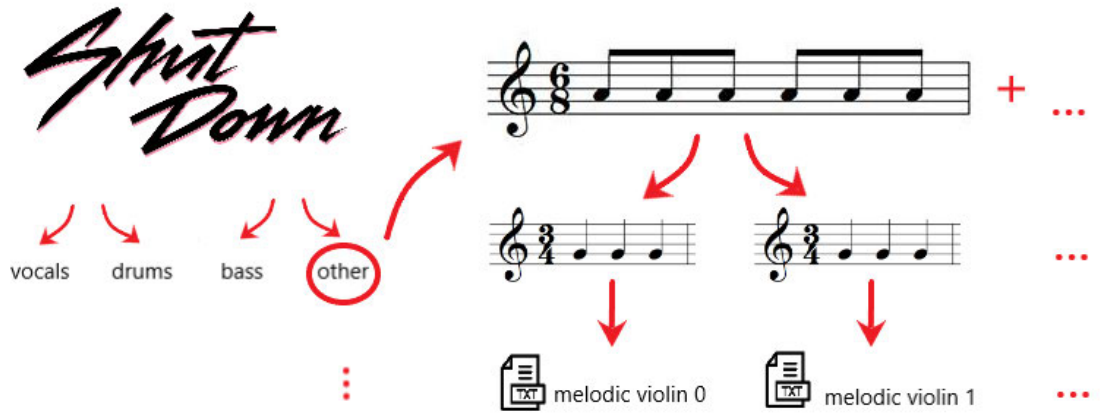



Figure 4.3.4.1: Shut Down model layout in initial configurations.

It should be noted that despite being a general rule, it has not been applied to all bars. Some of them had black or white notes that made the task easier from the beginning, other times the presence of sixteenth notes forced a new subdivision of the compass, etc... But in most cases the arrangement presented in figure 4.3.4.1 was the default option when searching with finder.py.

I understand that the organization of the model in so many python files is confusing, I had to face that confusion forced by circumstances. Each track of the song had to be modeled separately due to internal OpenGL settings. The library is only capable of working on one popup window at a time, that is, there is a global python variable called activeWindow that represents the window on which it works, even if the program has created several.

To solve this problem, thanks Jaime, the solution was quite sophisticated: Create four independent mirror python environments and each only work with one popup. If each virtual environment is in charge of a window, ultimately it must also be in charge of its corresponding track, which leads me to divide the model into several python files. Taking this into account, the execution of the different files would be carried out like this:

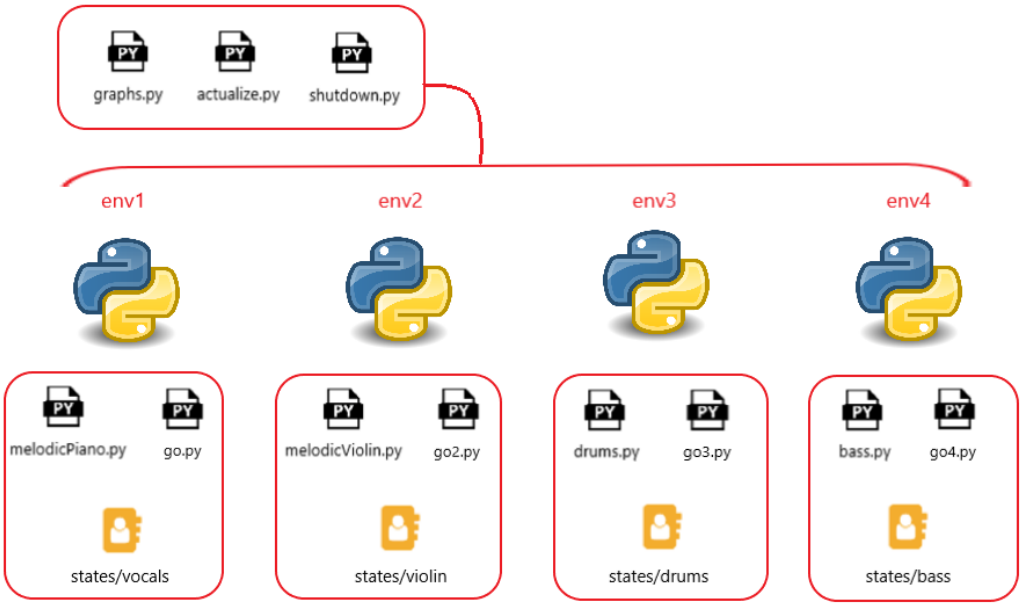



Figure 4.3.4.2: Diagram of model's execution with 4 virtual environments

Still, this is like the mole game. Once one problem is fixed, another appears on another site. No matter how fast the computer is, simultaneously executing 4 python files that simultaneously execute a changing graphic representation and a constant background sound are never unitary operations. Controlling the tempo is crucial here. Although the array handling operations are very fast, they will be carried out 100x100 in scaling and this is symmetrical for reading each configuration saved as a txt file.

It must also be taken into account that the graphical operations that draw living cells do their job only if the cells in question are alive indeed. The time it takes to graph each state will vary according to the number of living cells it has, it will not be a constant value. While this lag cannot be avoided, the random seed that generates the random states assumes a 50/50 ratio for dead and alive. Although some states will be drawn faster and others slower, in proportion it should not be too noticeable in the final result.

**go.py**

```
#import graphs as g
import melodicPiano as mp
import shutdown as s
import threading

### THREADING APPROACH
t1 = threading.Thread(target=mp.main,args=(375,))
t1.start()
t2 = threading.Thread(target=s.piano)
t2.start()

### NO-THREADING APPROACH
mp.main2()
```

The above code snippet renders the skeleton for each go.py launcher of each model track. It starts by importing the python file corresponding to the track and the common shutdown.py file where the full song is located. And it ends by launching two threads in the threading case and only one in the other case.

On the other hand, each python file dedicated to the corresponding tracks presents a different skeleton, taking the vocals of the song as an example:

**melodicPiano.py**

```
import graphs as g
import actualize as a
from OpenGL.GL import *
from OpenGL.GLU import *
from OpenGL.GLUT import *
import random
import time
import numpy as nm

import fluidsynth as fs
from mingus.containers import Note
sounds = "\yamahap45.sf2"
n= 0.0600
```

The common libraries graphs.py and actualice.py are imported, containing the adapted graphing operations and the rules of the game of life respectively. For the same purpose as in other files, the time, numpy and random libraries are also imported since they will be equally relevant here, as well as fluidsynth which resumes its main role as sound player.

For the threading case, the function that is called from go.py would be the following:

```
### THREADING APPROACH
def main(states):

### Initial declarations and global variables
    cellColor = g.PINK
    name = "Vocal melodies"
    posX = 0
    posY = 0

    ### Big Bang
    universe = [[ 0  for i in range(g.dimXgrid)] for j in range(g.dimYgrid)]
    universe = [[ random.choice([0,0,0,1,1,1]) for i in range(g.dimXgrid)] for j in range(g.dimYgrid)]
    speed=0.18

    # OpenGL actions carried once per universe
    glutInit()
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB)
    glutInitWindowPosition(posX,posY)
    glutInitWindowSize(g.dimXwindow,g.dimYwindow)
    glutCreateWindow(name)

    g.init()
    g.grid()

    # Loop
    for x in range(states):
     #glutSetWindow(glutGetWindow())
     glClear(GL_COLOR_BUFFER_BIT)

     g.drawLivings(cellColor,universe)
     time.sleep(speed)

     universe = a.nextState(universe)
```

As it can be seen, it is a procedural and simplified version of the yawnoc.py structure. With the necessary initial variables and declarations, a set of OpenGL actions that are only performed once, and another set that is repeated in a loop. The loop this time does not need to be a GLUT loop, since the duration and the number of states to draw are known. This allows the implementation of a `for` loop that is much more understandable than its predecessor.

Without giving it too much importance, since the values have been extracted in a completely subjective way, the tempo of each track varies slightly since they do not all perform the same number of operations.

| Threading | No Threading | | | |
|---|---|---|---|---|
| n= 0.09 | vocals(piano): n = 0.077 | drums: n ≃ 0.062 | violin: n ≃ 0.063 | bass: n ≃ 0.08 |

Figure 4.3.4.3: Value for n (duration of the eighth note) for each track

The file would conclude with the function of the second approach dedicated to avoiding threading. Given its length and the fact that it literally replicates its corresponding snippet of the song (which is already fully modeled in shutdown.py), it won't be included here. Instead, I'll continue the section directly with the file mentioned.

**shutdown.py**

```
import time
import fluidsynth as fs
from mingus.containers import Note
import  pygame
from pygame import mixer

n = 0.09

sounds = "\yamahap45.sf2"
sounds3 = "\drums"
sounds4 = "\R-violin.sf2"
sounds5 = "\TEK bass.sf2"
sounds6 = "\pizzicato2.sf2"
```

Both audio libraries are required along with the mingus notation relating to music theory. On the other hand, it is required to declare the relative path of each instrument, so that they are used when necessary.

And from here on, what remains is nothing more than the Shut Down model itself, bar by bar. Assuming some intermediate playback loop that does not require additional feedback, and starting with the piano, the model takes the following form:

**Shut Down vocals:**

The environment is created with the classical piano.

```
def piano():
    s = fs.Synth(gain=1.8)
    s.start()

    ins = "..\soundfonts"+ sounds
    soundfont = s.sfload(ins)
    s.program_select(0,soundfont,0,0)
```

The string intro that kicks off the song, the intro choruses that bring back the group's signature, and Jennie/ Lisa/ Jisoo/ Rosé first verses are modeled.

```
    # Initial Sostenuto + opening strings
    opening = [[int(Note("F", 5)),int(Note("Bb", 5)),int(Note("Bb", 5))],[int(Note("F", 6)),int(Note("F", 6)),int(Note("Eb", 6))],
    [int(Note("Db", 6)),int(Note("Db", 6)),int(Note("C", 6))],[int(Note("Bb", 5)),int(Note("A", 5))]]

    opSpeed = [[8,4,2],[2,2,2],[2,2,2],[2,2]]

    # BLACKPINK in ur area
    barsChants = [[int(Note("Db", 5))],[int(Note("F", 5)),int(Note("C", 5)),int(Note("C", 5))],
    [int(Note("C", 5)),int(Note("C", 5)),int(Note("F", 5))],[int(Note("Ab", 5))],[int(Note("F", 5))]]

    speedChants = [[2],[2,2,2],
    [1,1,14],[2],[12]]

    # Jennie
    bars1 = [[int(Note("Db", 5)),int(Note("Db", 5)),int(Note("Db", 5)),int(Note("Db", 5)),int(Note("Db", 5))],
    [int(Note("Db", 5)),int(Note("Bb", 4)),int(Note("Bb", 4))],[int(Note("C", 5)),int(Note("C", 5)),int(Note("C", 5))],
    [int(Note("F", 5))]]

    speed1 = [[2,1,1,1,1],
    [2,2,2],[2,2,2],
    [6]]

    bars2 = [[int(Note("Db", 5)),int(Note("Db", 5)),int(Note("Db", 5)),int(Note("Db", 5)),int(Note("Db", 5)),int(Note("Db", 5))],
    [int(Note("F", 5)),int(Note("Eb", 5)),int(Note("Eb", 5))],[int(Note("Eb", 5)),int(Note("Eb", 5)),int(Note("Db", 5))],
    [int(Note("Db", 5)),int(Note("Eb", 5))],[int(Note("Db", 5)),int(Note("Db", 5)),int(Note("Db", 5)),int(Note("F", 4))],
    [int(Note("C", 5)),int(Note("C", 5)),int(Note("C", 5)),int(Note("C", 5))]]

    speed2 = [[1,1,1,1,1,1],
    [2,2,2],[2,2,2],
    [1,5],[1,1,2,8],
    [2,2,2,6]]
```

```
    # Lisa
    bars3 = [[int(Note("Db", 5)),int(Note("Db", 5)),int(Note("Db", 5)),int(Note("Db", 5)),int(Note("Db", 5)),int(Note("Db", 5))],
    [int(Note("F", 5)),int(Note("F", 5)),int(Note("Eb", 5)),int(Note("Eb", 5))],[int(Note("Eb", 5)),int(Note("Eb", 5)),int(Note("Db", 5))
    [int(Note("Db", 5)),int(Note("Eb", 5))],[int(Note("Db", 5)),int(Note("Db", 5)),int(Note("Db", 5)),int(Note("F", 4))],
    [int(Note("C", 5)),int(Note("C", 5)),int(Note("F", 5)),int(Note("Bb", 4))]]

    speed3 = [[1,1,1,1,1,1],
    [1,1,2,2],[2,2,2],
    [1,5],[1,1,2,8],
    [2,2,2,6]]
```

```
    # Jisoo
    bars1 = [[int(Note("F", 5)),int(Note("F", 5)),int(Note("F", 5)),int(Note("F", 5)),int(Note("F", 5)),int(Note("F", 5))],
    [int(Note("F", 5)),int(Note("F", 5)),int(Note("F", 5)),int(Note("F", 5)),int(Note("F", 5))],
    [int(Note("Ab", 5)),int(Note("F", 5)),int(Note("F", 5))]]

    speed1 = [[1,1,1,1,1,1],
    [1,1,1,1,2],
    [4,2,6]]

    # Rosé
    bars2 = [[int(Note("F", 5)),int(Note("F", 5)),int(Note("F", 5)),int(Note("F", 5)),int(Note("F", 5))],
    [int(Note("F", 5)),int(Note("F", 5)),int(Note("F", 5)),int(Note("F", 5))],
    [int(Note("Ab", 5)),int(Note("Bb", 5)),int(Note("F", 5))]]

    speed2 = [[1,1,1,1,2],
    [2,1,1,2],
    [4,2,6]]

    bars3 = [[int(Note("F", 5)),int(Note("F", 5)),int(Note("F", 5)),int(Note("F", 5)),int(Note("F", 5)),int(Note("F", 5))],
    [int(Note("F", 5)),int(Note("F", 5)),int(Note("F", 5)),int(Note("F", 5)),int(Note("F", 5))],
    [int(Note("F", 5)),int(Note("Eb", 5)),int(Note("Db", 5))],[int(Note("Bb", 4))]]

    speed3 = [[1,1,1,1,1,1],
    [1,1,1,1,2],
    [2,2,2],[2]]
```

And finally, the main melody of the chorus and the two versions of the middle section "Whip it, Whip it, Whip it" are modeled.

```
    # Chorus vocal lines
    bars1 = [[int(Note("Bb", 4)),int(Note("Bb", 4))],[int(Note("Bb", 4)),int(Note("Bb", 4)),int(Note("Bb", 4))],
    [int(Note("Bb", 4)),int(Note("Bb", 4)),int(Note("Db", 5))],[int(Note("F", 5)),int(Note("C", 5))]]

    speed1 = [[2,2],[2,2,2],
    [2,2,2],[2,6]]

```

```
    # Whip it,Whip it, Whip it version 1
    bars2 = [[int(Note("F", 5)),int(Note("F", 5)),int(Note("Eb", 5)),int(Note("Eb", 5)),int(Note("Eb", 5)),int(Note("Eb", 5))],
    [int(Note("F", 5)),int(Note("F", 5))],
    [int(Note("F", 5)),int(Note("F", 5)),int(Note("Eb", 5)),int(Note("Eb", 5)),int(Note("Eb", 5)),int(Note("Eb", 5))],
    [int(Note("F", 5)),int(Note("F", 5)),int(Note("Bb", 4)),int(Note("Bb", 4)),int(Note("Bb", 4))],
    [int(Note("Bb", 4)),int(Note("Bb", 4)),int(Note("Bb", 4))],[int(Note("Db", 5)),int(Note("F", 5)),int(Note("C", 5))]]

    speed2 = [[1,1,1,1,1,1],
    [1,5],
    [1,1,1,1,1,1],
    [1,3,2,2,2],
    [2,2,2],[2,2,6]]

    # Whip it, Whip it, Whip it version 2
    bars3 = [[int(Note("F", 5)),int(Note("F", 5)),int(Note("Eb", 5)),int(Note("Eb", 5)),int(Note("Eb", 5)),int(Note("Eb", 5))],
    [int(Note("F", 5)),int(Note("F", 5))],
    [int(Note("F", 5)),int(Note("F", 5)),int(Note("Eb", 5)),int(Note("Eb", 5)),int(Note("Eb", 5)),int(Note("Eb", 5))],
    [int(Note("F", 5)),int(Note("F", 5)),int(Note("F", 5)),int(Note("Gb", 5))],
    [int(Note("F", 5)),int(Note("Eb", 5)),int(Note("Db", 5)),int(Note("C", 5)),int(Note("Bb", 4))]]

    speed3 = [[1,1,1,1,1,1],
    [1,5],
    [1,1,1,1,1,1],
    [1,5,2,2,2],
    [2,2,2,2,4]]
```

**Shut Down bass:**

The environment is created with the deep-trap bass related to the 808.

```
def bassFS ():
    s = fs.Synth(gain=0.8)
    s.start()

    ins = "..\soundfonts"+ sounds5
    soundfont = s.sfload(ins)
    s.program_select(0,soundfont,0,0)
```

And showing off the harmonic simplicity of the song, Shutdown's bass is modeled from the intro and the chorus. Although the song's outro incorporates a third chord, this is beyond the scope of the current model.

```
    # Trap bass chords
    BbmChord = [int(Note("Bb", 2)),int(Note("Db", 3)),int(Note("F", 3))]
    FChord = [int(Note("F", 3)),int(Note("Ab", 3)),int(Note("C", 4))]

    bars = [BbmChord,[0,0,0],FChord,FChord,FChord,BbmChord,[0,0,0],FChord,FChord]

    speed = [6,6,2,8,2,6,6,2,10]
```

**Shut Down others (violin instrumental):**

The environment is created with the classical violin.

```
def violin():
    s = fs.Synth(gain=0.1)
    s.start()

    ins = "..\soundfonts"+ sounds4
    soundfont = s.sfload(ins)
    s.program_select(0,soundfont,0,0)
```

The main motif of the sample "La Campanella" and the initial sostenuto that begins the piece are modeled.

```
    # Initial Sostenuto + opening strings
    opening = [[int(Note("F", 5)),int(Note("Bb", 5)),int(Note("Bb", 5))],[int(Note("F", 6)),int(Note("F", 6)),int(Note("Eb", 6))]]

    opSpeed = [[8,4,2],[2,2,2]]

    # Leitmotiv: "La Campanella"
    bars = [[int(Note("Db", 6)),int(Note("Db", 6)),int(Note("C", 6))],[int(Note("Bb", 5)),int(Note("A", 5)),int(Note("Bb", 5))],
    [int(Note("C", 6)),int(Note("F", 5)),int(Note("F", 5))],[int(Note("Gb", 5)),int(Note("F", 5)),int(Note("Eb", 5))]]

    speed = [[2,2,2],[2,2,2],[2,2,2],[2,2,2]]
```

The environment is created with the classical violin in pizzicato mode and the bars dedicated to Paganini's pizzicatos are modeled, which Teddy recovers in the pre-chorus.

```
def violinPizzicatos():

    s = fs.Synth(gain=0.3)
    s.start()

    ins = "..\soundfonts"+ sounds6
    soundfont = s.sfload(ins)
    s.program_select(0,soundfont,0,45)

    # Paganini's Pizzicatos
    bars1 = [[int(Note("Bb", 4)),int(Note("Bb", 4)),int(Note("Db", 5))],[int(Note("F", 5)),int(Note("Db", 5)),int(Note("Bb", 4))]]

    bars2 = [[int(Note("A", 4)),int(Note("Ab", 4)),int(Note("C", 5))],[int(Note("F", 5)),int(Note("C", 5)),int(Note("Ab", 4))]]

    speed = [[2,2,2],[2,2,2]]
```

And finally, the ending resource with the classic i-V-I (the fifth grade in second inversion) is modeled with which the chorus is concluded.

```
    #  Termination: i-V-i (V in second inversion)
    outro = [[int(Note("F", 5)),int(Note("Bb", 5)),int(Note("Db", 6)),int(Note("F", 6))],
    [int(Note("F", 5)),int(Note("A", 5)),int(Note("C", 6)),int(Note("F", 6))],
    [int(Note("Bb", 4)),int(Note("Db", 5)),int(Note("F", 5))]]

    ouSpeed =  [[3,1,1,1],[3,1,1,1],[8,8,8]]
```

**Shut Down drums:**

The environment is created with the Roland TR-808 drum set and its samples.

```
def drums():
    pygame.init()

    ins = "..\soundfonts"+ sounds3
    bassDrum = mixer.Sound(ins+"\BassDrum.WAV")
    bassDrum.set_volume(0.5)
    snareDrum = mixer.Sound(ins+"\SnareDrum.WAV")
    snareDrum.set_volume(0.5)
    hihat = mixer.Sound(ins+"\Hihat.WAV")
    hihat.set_volume(0.5)
    finalDrum = mixer.Sound(ins+"\FinalDrum.WAV")
    finalDrum.set_volume(0.5)
```

The rhythmic patterns present in the first verse are modeled. Resulting almost identical except that one incorporates the TUM (finalDrum) and the other does not.

```
    # Standard rhytmic pattern
    pattern = [[bassDrum,hihat,hihat,snareDrum,hihat,hihat,hihat,hihat,bassDrum,bassDrum,snareDrum,hihat,bassDrum,hihat,hihat],
    [bassDrum,hihat,hihat,snareDrum,hihat,hihat,hihat,bassDrum,hihat,bassDrum,snareDrum,hihat,snareDrum]]

    # TUM rhytmic pattern
    patternTum =[[bassDrum,hihat,hihat,snareDrum,hihat,hihat,hihat,hihat,bassDrum,bassDrum,snareDrum,hihat,bassDrum,hihat,hihat],
    [bassDrum,hihat,hihat,finalDrum,hihat,hihat,hihat,bassDrum,hihat,bassDrum,snareDrum,hihat,snareDrum]]

    speed = [[3,3,3,3,3,1,1,4,3,3,3,3,1,1,1],
    [3,3,3,3,3,3,3,1,2,3,3,3,3]]
```

And finally, the rhythmic patterns of the pizzicatos section and the chorus are modeled respectively. Interesting that during the pizzicatos bassDrum hits are continuously avoided and that during the chorus the frequency of the TUM (finalDrum) increases considerably.

```
    # Standard pattern during pizzicatos (bassDrum OUT/ hihat IN)
    pattern = [[hihat,hihat,hihat,snareDrum,hihat,hihat,hihat,hihat,hihat,snareDrum,hihat,hihat,hihat,hihat],
    [hihat,hihat,hihat,snareDrum,hihat,hihat,hihat,hihat,hihat,hihat,snareDrum,hihat,snareDrum],
    [hihat,hihat,hihat,snareDrum,hihat,hihat,hihat,hihat,hihat,snareDrum,hihat,hihat,hihat,hihat],
    [hihat,hihat,hihat,snareDrum,hihat,hihat]]

    speed = [[3,3,3,3,3,2,4,3,3,3,3,1,1,1],
    [3,3,3,3,3,3,3,1,2,3,3,3,3],
    [3,3,3,3,3,2,4,3,3,3,3,1,1,1],
    [3,3,3,3,3,3]]
```

```
    # Standard Chorus rhytmic pattern
    pattern = [[bassDrum,hihat,hihat,snareDrum,hihat,hihat,bassDrum,bassDrum,finalDrum,hihat,bassDrum,hihat,hihat],
    [bassDrum,hihat,hihat,snareDrum,hihat,bassDrum,hihat,bassDrum,finalDrum,hihat,snareDrum],
    [bassDrum,hihat,hihat,snareDrum,hihat,hihat,hihat,hihat,bassDrum,bassDrum,snareDrum,hihat,bassDrum,hihat],
    [bassDrum,hihat,hihat,snareDrum,hihat,hihat,hihat,bassDrum,hihat,bassDrum,snareDrum,hihat,snareDrum],
    [bassDrum,hihat,hihat,snareDrum,hihat,hihat,bassDrum,bassDrum,finalDrum,hihat,bassDrum,hihat,hihat],
    [bassDrum,hihat,hihat,snareDrum,hihat,hihat,bassDrum,hihat,bassDrum,finalDrum,hihat,snareDrum],
    [bassDrum,hihat,hihat,snareDrum,hihat,hihat,hihat,hihat,bassDrum,bassDrum,snareDrum,hihat,bassDrum,hihat,hihat]]

    speed = [[3,3,3,3,3,6,3,3,3,3,1,1,1],
    [3,3,3,6,6,1,2,3,3,3,3],
    [3,3,3,3,3,3,1,1,4,3,3,3,3,2,1],
    [3,3,3,3,3,3,3,3,1,2,3,3,3,3],
    [3,3,3,3,3,6,3,3,3,3,1,1,1],
    [3,3,3,3,3,6,1,2,3,3,3,3],
    [3,3,3,3,3,3,1,1,4,3,3,3,3,1,1,1]]
```

To conclude section 4, dedicated to presenting the sound and possibilities of Conway's game of life, an image of the internal structure of Shut Down and a capture of all the tracks playing simultaneously together have been included.

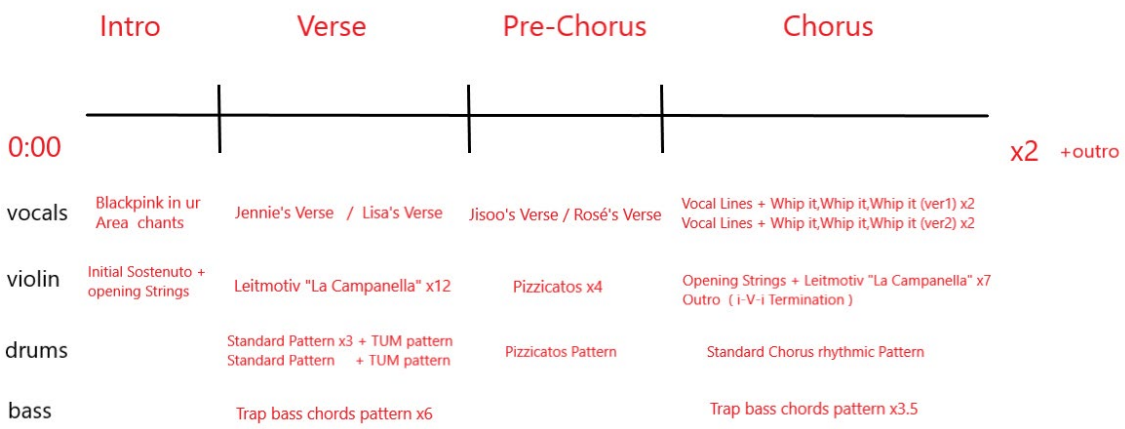



Figure 4.3.4.4: Shut Down structure and each track's behaviour

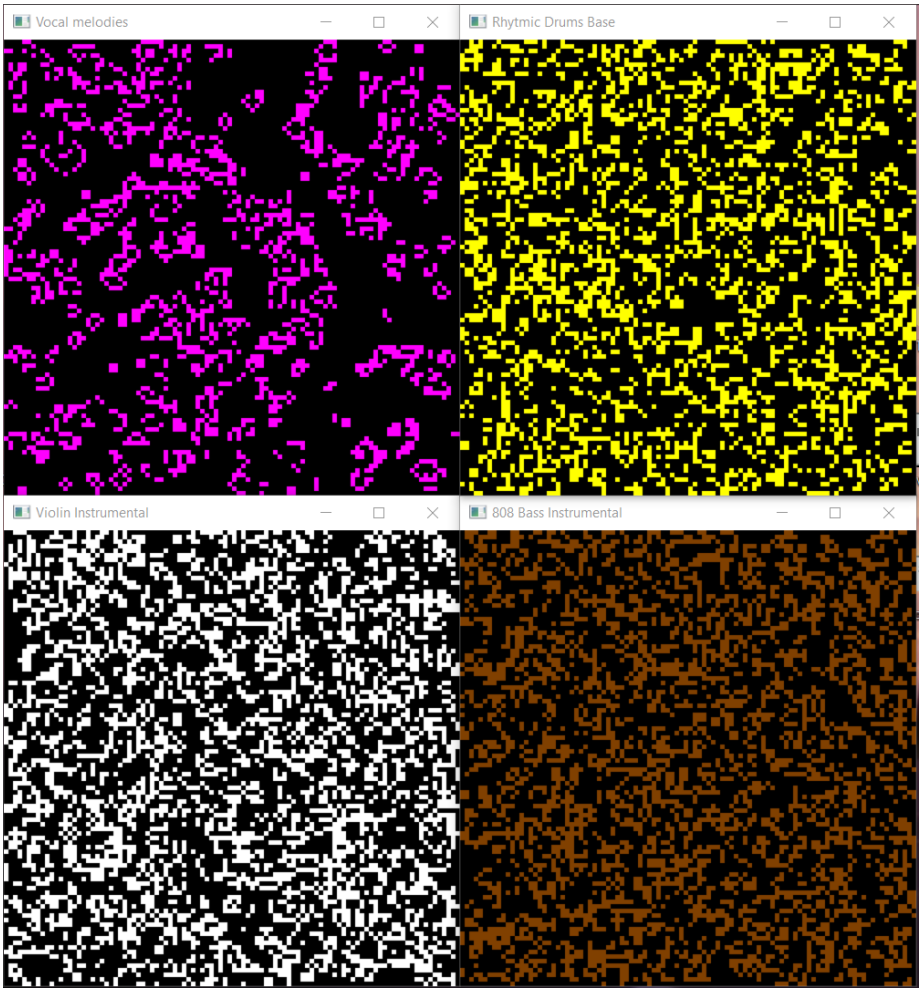



Figure 4.3.4.5: Simultaneous running of all four Shut Down tracks.

# 5. PLANNING AND METHODOLOGY

As was superficially commented in section 3, the project was carried out following the guidelines of an agile methodology based on work phases with several milestones. In short, each dedicated to the implementation and verification of one or several functional requirements, until a high priority requirement was correctly codified and tested. The following schedule shows the time distribution of each phase, as well as the requirements that were completed at each moment. With the intention of completing the project within the estimated times, an average of 40 hours of work per week is assumed.

| | NOVEMBER | | DECEMBER | | | | JANUARY | | | | FEBRUARY | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | W3 | W4 | W1 | W2 | W3 | W4 | W1 | W2 | W3 | W4 | W1 | W2 |
| **PHASE 1: PLANNING, RESEARCH AND VERIFICATION ACTIONS** | | | | | | | | | | | | |
| Brainstorming, distribution of work in time, specification of requirements and general planning. | | | NFR 02 | | | | | | | | | |
| Research, study of the context, applicability and tools necessary to achieve the milestone. | | | | | | | | | | | | |
| Unit tests, introduction of parameters or modular tests. | | | | | | | | | | | | |
| **PHASE 2: ACTIONS RELATED TO THE CONWAY'S GAME OF LIFE IN ITS TRADITIONAL VERSION** | | | | | | | | | | | | |
| Visual display of a state of the game of life on screen. | | | FR 01 | FR 06 | | FR 04,5 | | | | | | |
| Notion of adjacency and update set of rules. | | | | FR 07 | | | | | | | | |
| Dynamic mode and runtime actions. | | | | | FR 02 03 | FR 08 09 | | | | | | |
| **PHASE 3: ACTIONS RELATED TO THE POSSIBLE APPLICATION AS A VISUALIZER OF GAME OF LIFE** | | | | | | | | | | | | |
| Association function between one given state and a value. | | | | | | | | | | | | |
| Association between a value and a sound. Playing that sound in the game of life. | | | | | | | | FR 10 | | | | |
| Model of a modern song in the game of life. | | | | | | | | | FR 11 | FR 12 | | |
| **PHASE 4: ACTIONS RELATED TO THE COMPLETION OF THE PROJECT AND THE REVIEW OF THE ARTIFACTS ARISING FROM IT.** | | | | | | | | | | | | |
| Reflect the conclusions of the project in the report and development of the user guide. | | | | | | | | | | | NFR 01 | |
| Preparation of the oral defense of the project | | | | | | | | | | | | |

# 6. SOCIAL AND ENVIRONMENTAL IMPACT: ETHICAL AND SOCIAL RESPONSIBILITY

Ironically, it would be unethical to rescue my own argument on this topic [16] without contributing with new nuances or horizons. I have tried to argue with myself and refute my ideas presented in the homonymous section of my first final degree project, but it has been impossible. It's been akin to playing chess with both the black and white pieces at the same time, there's no arguing if I'm still subscribing to every word I left behind.

The reason for being an engineer lies in providing solutions to very real problems within a specific sector, not only acting as a simple creator, but also self-attributing the role of promoter of change. This is not trivial, much less free: you have a gift but this comes with a price. And the bitter truth is that you can never know the outcome a priori not even the price you will have to pay. One of those small splinters, which unfortunately many engineers choose to ignore, or worse, which they were not even aware of, is social responsibility.

Knowledge is undoubtedly an unquestionable advantage, to such an extent that it allows certain individuals to perform certain actions while preventing others from doing so. This small injustice of nature, intrinsic to the impossibility of knowing everything, must be corrected by contributing positively. What others cannot do but still need becomes your responsibility and vice versa. Once again, I regret being part of an often corrupt system based on results, optimization and profit maximization, as this ends up distorting that much-needed balance, transforming it into a technological, social, intellectual and generational gap.

Irrefutable proof of this can be found in this project, where a not very veiled criticism of the engineer's ethics coexists and at the same time with a work that lacks measurable applications towards any social problem. It is not an excuse that my intentions were not perverse, because they were quite selfish. I wanted to dedicate my work to music, mathematics and software with the intention of enjoying the last bits of my university life before facing the next stage.

It is true that showing music in a visual and simpler way can be invaluable for people with a disorder that prevents them from hearing correctly. It is also true that trying to spread the word about a classic programming problem and looking for other applications can also be beneficial. It wouldn't even be unfair to admit that the project ended gamifying the music and recovering the undeniable influence of classical music in modern production. But all this is irrelevant in light of the fact that I did not do the project to achieve any of this. As much as some noteworthy implications may be drawn, the key to morality, at least from a Kantian point of view, lies in the intention. And my intentions were not far from the illusion of a child who discovers a new game and suffers from synesthesia, a child who wanted to get away from the problems of adult life by listening to BLACKPINK.

As in other similar situations and other precedents, the only solution to avoid frustration and not feel guilty is to embrace nihilism. By remembering that it is nothing more than a simple final degree project, that the scope of my project would never have reached the point of being strictly key to anything at all and that this trip has been just a necessary procedure to certify my college degree.

But there is hope, guilt as a tool. The simple fact that I am aware of my mistakes will condition and slightly bias my future decisions, a privilege that others do not have now and will most certainly never have.

# 7. CONCLUSIONS AND DEVELOPMENT PROPOSALS

The overall feeling is quite satisfying. Although to my knowledge not all the objectives were achieved, the result has met both the usual quality standards and the somewhat high expectations of this perfectionist engineer. The time estimates devised in section 5 were met without exception and despite compressing the time available to carry out the project in a few months, the sensations danced between the illusion of a child when he receives a gift and the unexpected satisfaction of working on something by vocation and not by obligation.

Despite, as I have said, the product is up to par, it is not as understandable as I would have liked, not much less accessible given the large number of dependencies and installations prior to the use of the application itself. As much as it annoys me, Python's understandability is not enough to make code easy to understand and manipulate if not done from a developer user's point of view.

Although the rest of the objectives were achieved, if my intention was to study the possible application of the game of life to other areas, only its possible role as a visualizer was particularly studied. The result is visually pleasing, but if you don't launch the game of life on a big enough board, it's downright hard to make out graphic patterns. As possible solutions to this event, perhaps the possibility of reducing the amount of living cells in the random generation of states is raised, reducing chaotic explosions and forcing the appearance of more suggestive oscillators and still lifes.

On the other hand, I find it interesting to highlight that despite not considering it as an objective at the beginning of the project, my ability to adapt and resolve contingencies was up to the task. During the execution of the project, many problems and eventualities arose unexpectedly that could have slowed down or affected the overall planning, but they were always solved within the established deadlines. Most of these problems have been discussed during the code explanation in the previous sections, displaying an excessive storytelling resource.

Most of the proposals or updates that I think would be interesting for yawnoc are additions of new features.

**Proposals**:

- A graphical user interface. All the python modules are within the backend sector of the project, but at no time were requirements for an interface specified. Perhaps one of the reasons why the product has not been so accessible is the absence of a frontend. The possibility of including a simple and accessible interface that facilitates the use of the application is raised, avoiding the entry of commands through the console.

- The inclusion of more musical instruments. Taking advantage of the graphical interface, it would be interesting to include more instruments and samples in .sf2 format. Allowing the user a more immersive and quality experience, without having to deal with so many restrictions and defects in the reproduction of the notes.

- Introduce more complex chords and other modes. Only the major and minor keys are included in the harmonic approach of the work, and for simplicity only triad chords were included in their fundamental state. Since mingus provides great support for this eventuality, the possibility of including the rest of the modes and more complex or inverted chords should not be ruled out.

- Allow the application to work and create files in midi format. The composition possibilities of the game of life are quite interesting. Working with midi files could open up a whole new world of possibilities when it comes to creating melodic/harmonic and rhythmic patterns.

- The search and inclusion of different association functions. I am sure that I have shown my rejection before the simplicity of the current association function presented in section 4. Although it was done prioritizing the update speed of each state over the complexity and depth of it, a more interesting and complex function for layers would give the game much more quality. For now, only basic combinatorics operations are performed with prime numbers and mathematical analysis only makes a timid appearance in the mutation phase. I leave to whoever may be interested, including the future version of myself, the mission of finding a function that is closer to the true nature of sound.

# 8. BIBILIOGRAPHY


[1] Wolfram, Stephen. (1st edition).(1983). *"Statistical mechanics of celular automata". (Rev.Mod.Phys. 55, 601) American Physical Society.*

[2] Wolfram, Stephen. (1st edition).(2002). *"A New Kind of Science". Wolfram Research.* Retrieved on December 16, 2022 from https://www.wolframscience.com/nks/

[3] López García, Guillermo (2022). "*Conway's Game of Life*". Final Degree Project, E.T.S. de Ingenieros Informáticos (UPM), Madrid, España. Retrieved on December 10, 2022 from https://oa.upm.es/71278/1/TFG_GUILLERMO_LOPEZ_GARCIA.pdf

[4] Lifewiki. *The wiki for Conway's game of life.* Retrieved on December 13, 2022 from https://conwaylife.com/wiki/Main_Page

[6] Game of Life. *Play the Game of Life online.* Retrieved on December 13, 2022 from https://playgameoflife.com/

[7] Damien Regnault, Nicolas Schabanel, Eric Thierry. (2010) "*On the analysis of "simple" 2D stochastic cellular automata*". Discrete Mathematics and Theoretical Computer Science, Vol. 12 no. 2 (2), pp.263- 294. Retrieved on December 17, 2022 from https://hal.inria.fr/hal-00990468

[8] Demucs. *Demucs Music Source Separation.* Github Repository cloned on December 18, 2022 from https://github.com/facebookresearch/demucs

[9] Carlini, Nicholas. (2020) "*Digital Logic gates on Conway's Game of Life – Part 1*". [Online] Retrieved on December 17, 2022 from https://nicholas.carlini.com/writing/2020/digital-logic-game-of-life.html

[10] Online Life-Like CA Soup Search. *Long-Lived Patterns in Conway's Life.* Retrieved on December 17, 2022 from https://web.archive.org/web/20101203183556/http://www.conwaylife.com/soup/methuselahs.asp?rule=B3/S23

[11] Hookheory. *Shut Down by Blackpink Chords and Melody.* Retrieved on December 17, 2022 from https://www.hooktheory.com/theorytab/view/blackpink/shut-down

[12] Musescore. *Shut Down – Blackpink piano Sheet.* Retrieved on December 17, 2022 from https://musescore.com/user/28856247/scores/8653641

[13] Rossini, Gabo. (2022) "*BLACKPINK – 'Shut Down' M/V Composer Reacts Music Analysis*". [Youtube] Retrieved on November 17, 2022 from https://www.youtube.com/watch?v=10lfv5Z-s2k

[14] Musescore. *Étude No. 3 in G♯ minor, "La Campanella"*. Retrieved on December 18, 2022 from https://musescore.com/classicman/scores/106022

[15] Musescore. *Black Pink – Shut Down Sheet Music (Preview).* Retrieved on December 27, 2022 from https://musescore.com/user/39818920/scores/8658117

[16] Carlés Durá, Luis (2022). "*La Función Zeta de Riemann y su Relación con los núemros primos*". Final Degree Project, E.T.S. de Ingeniería de Sistemas Informáticos (UPM), Madrid, España. Retrieved on December 10, 2022 from https://oa.upm.es/71576/

[17] Vaswani Ashish, Shazeer Noam, Palmar Niki, Uszkoreit Jakob, Jones Llion, Gomez Aidan N., Kaiser Lukasz, Polosukhin Illia . (2017) "*Attention is all you need*". NIPS'17: Proceedings of the 31st International Conference on Neural Information Processing Systems. Curran Associates Inc. Retrieved on December 27, 2022 from https://arxiv.org/abs/1706.03762

[18] Mingus. *Python Documentation for mingus package.* Retrieved on January 04, 2023 from https://bspaans.github.io/python-mingus/index.html

[19] FluidSynth. *Playing music from python via fluidsynth.* Retrieved on January 05, 2023 from https://ksvi.mff.cuni.cz/~dingle/2019/prog_1/python_music.html

[20] Pyfluidsynth error. *Fluidsynth: error: unknown integer parameter'synth.sample-rate'*. Retrieved on January 06, 2023 from https://github.com/nwhitehead/pyfluidsynth/issues/37

[21] Wikipedia.org . *Ulam Spiral.* Retrieved on January 12, 2023 from https://en.wikipedia.org/wiki/Ulam_spiral

[22] yawnoc. *Final Degree Project: Conway's Game of Life Implementation and its application as a visualizer.* Github Repository cloned on February 08, 2023 from https://github.com/Luis-Carles/yawnoc.git