Jan-Øivind Lima

# Iterative Development of a Constraint-Based Procedural Music Generator

From simple melodic rules to form- and harmonically-aware generation

Master's thesis in Cybernetics and Robotics
Supervisor: Sverre Hendseth
June 2026

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

NTNU

# ABSTRACT

This thesis investigates how a procedural music generator for tonal melodies can be developed through iterative software design. The system was built in three iterations, progressing from a simple proof of concept to a more structured generator with explicit musical objects, form handling, harmonic planning and soft constraints. The final system can generate both single-line melodies and chorale-style outputs from either automatically derived or explicitly specified harmonic context. The development process also suggests that artificial intelligence can be a useful aid for coding tasks if used as a tool to augment the engineers' abilities rather than replacing them.

# SAMMENDRAG

Denne oppgaven undersøker hvordan en prosedyregenerert musikkgenerator for tonale melodier kan bli utviklet gjennom iterativ programvaredesign. Systemet ble bygget i tre iterasjoner, og utviklet seg fra en enkel konseptutprøving til en mer strukturert generator med eksplisitte musikalske objekter, formhåndtering, harmonisk planlegging og myke restriksjoner. Det endelig systemet kan generere både enstemmige melodier og korallignende satser, enten fra automatisk utleder eller eksplisitt angitt harmonisk kontekst. Utviklingsprosessen tyder også på at kunstig intelligens kan være et nyttig verkøy for kodingsoppgaver hvis det brukes som et verktøy for å øke ingeniørens evner, heller enn å erstatte dem.

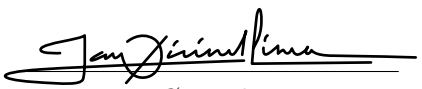
# PREFACE

This project is the culmination of my studies at Norwegian University of Science and Technology (NTNU) in the field of Cybernetics and Robotics. The project has been a long journey of learning and discovery, and I am grateful for the opportunity to explore the intersection of music, procedural generation, and artificial intelligence. This thesis was written during the spring of 2026. I would like to thank my supervisor, Sverre Hendseth, both for his guidance and support throughout the project, but also long anecdotes and stories which has been food for thought throughout the semester.

I would like to thank NTNUI Calisthenics for filling my spare time with great people and activities as well as being a place where I could be myself completely. Thanks to Trondheim Studentersangforening and UKEkoret Pirum for re-igniting my passion for music and giving me unforgettable memories from my time at NTNU.

Lastly I would like to thank my trusted lab-partner and good friend, Mathilde Øhra Levy. Without her, my five years at NTNU would not have been as enjoyable. Thanks for making projects, labs and assignments fun, no matter what our results were. Thanks for always reminding me to actually read the safety section of the lab manuals...

Trondheim, June 2026



*Jan-Øivind Lima*

# CONTENTS

# LIST OF FIGURES

# LIST OF EXAMPLES



All examples can be found at my github.io page where they can be listened to, numberings are consistent with the thesis.

CHAPTER

# ONE

# INTRODUCTION

For hundreds of years composers have been creating music using external tools like dice to help them generate something new. As early as the 18th century the idea of using randomness to help the composing process has been used. The musical dice game, "Musikalisches Würfelspiel", was used to randomly generate music based on pre composed parts by rolling a set of dice [1]. The most popular example of this is the *Musikalisches Würfelspiel* by Mozart, which was published in 1787 [2]. This is a very basic way of generating music, however by the random nature of the dice it will not be able to create a larger overarching musical idea. The output is highly dependent on the pre composed parts made by the author.

If we want to generate everything from scratch we need a way more sophisticated system. This is where procedural generation comes into the picture. If we base the melody on something random and then use the conventions and "rules" for melodic composition we might be able to generate music that sounds somewhat coherent.

Getting external help has not only been used by composers, but also by programmers. In recent years the use of AI in software development has become more and more widespread. As an experiment in this thesis I will compare "old-school" development and AI. I will start with normal development without using AI. Then after I have built up some knowledge of the domain and system I will try to improve upon it using AI tools and evaluate both the output of the code and the code itself.

## 1.1 Context and motivation

Procedural Content Generation (PCG) is today widespread and used in many different applications. Generating content for video games is a common example of PCG being used in an artistic process. However the use of content using Procedural Generation (PG) is not straightforward, there is a big difference between hand crafting content and automatically generated content. PCG enables us to create more content faster, however it will be at the expense of the quality unless we have a highly sophisticated generator [3]. However the use of PG is

not only for artistic purposes, it can have many other uses, a few examples are city planning and simulating vascular trees for medicinal purposes. The program *ArcGIS CityEngine* is an example of PG being used for assisting city planners in creating sketches, ideas or finished plans for cities [4]. And *VascuSynth* is used to generate volumetric image data used for vascular health analysis [5].

Today a very common way to go about generating music is using AI and Large Language Models (LLM), but what goes on inside these large models is not easy to understand and tweaking them for a specific use case is not trivial. Several state of the art music generation models are using these kinds of models to generate the music [6, 7, 8, 9, 10, 11] Using such models enables anyone to create music, however these models are trained on large data sets and imitates what has come before not generating something truly new. For this thesis the focus is on the craft of creating music, breaking it down to its very primitive form, only pitches and durations. This will enable the user to have more control over exactly what is being generated. However this requires us to create a system and models that capture the essence of how exactly we are to go about constructing the very basic musical ideas. It will not depend on large data sets and will create a musical ideas that can be processed into a fully fledged musical idea. This task is not trivial since the rules of creating music is not absolute and the innate human element of creativity needs to be done by a computer.

If we are able to solve this problem of creating a good structure and model for this kind of generation we will not only have a solution for musical ideas, but this kind of system could solve many kinds of creative problems.

## 1.2 Problem description

The problem we want to solve is the problem of creating a coherent system that will be able to generate music that adheres to certain musical principles. In order to do this we need to consider many things, and system design is a big part of this. When building such a program we need to consider what parts it should consist of as well as how they are supposed to interact. Object Oriented Programming (OOP) is a solid way to approach this system design. Additionally I will explore how the use of AI in software development can aid in solving problems and speeding up the process.
The main goals of the project are as follows:

- Develop a procedural music generator based on music theory principles
- Use iterative development to refine both the software design and the musical output
- Investigate how AI can be used as a software development tool

## 1.3 Problem solution

The ideal solution to this problem would be a system that not only is able to solve this problem of creating music that is self similar and coherent, but is constructed

in such a way that the solution can easily be generalized in order to solve other problems. The problem of generating something coherent is not special to only music, but has applications for many other domains, especially creative ones or where there are a large number of usable solutions. For instance ways to connect cities by roads.

## 1.4 Thesis structure

The thesis is structured as follows. Firstly there is some background to give some context. This background includes music theory, this section is just to give the reader a basic insight in how music is constructed and how notation works in order to understand figures and examples that show up throughout the thesis. Following is a section about procedural generation, this section explains a few methods of procedural generation briefly to give context about how a few methods work as well as some illustrative examples on how they can be utilized. After this there are a few brief sections about object oriented programming and the use of AI in software development. Then comes the bulk of the thesis with the iterations of the software. The three iterations are presented as their own complete system in separate chapters. Before ending the thesis with a discussion and the concluding thoughts.

## 1.5 Note on examples and figures

Throughout the thesis you can find both figure and examples. Examples are a special kind of figure which can be listened to in the example archive. All examples contain a link to the given example in the archive. Some figures also contain musical notation, but these are not meant to be listened to, but rather just to illustrate a point about the structure of the music.

CHAPTER

# TWO

# BACKGROUND

This chapter provides the background theory for the project. First we give an overview of procedural generation and some of the different methods used in procedural generation. Then we give an overview of some basic music theory that is necessary to fully grasp the structure of the system. Lastly we cover some about object oriented programming.

## 2.1 Procedural generation

PCG, often referred to as just PG, is algorithmic generation of content. This content can take many forms including landscape, levels, textures or even 3D models. This leads to smaller file sizes since the program can generate the content using algorithms rather than having everything pre-made and saved. In video games procedural generation is often used to create large spaces and to create high replayability. In the game *No Man's Sky* procedural generation was used to procedurally generate "infinite procedurally generated galaxy" [12]. In Figure 2.1.2 we can see some examples of the worlds and animals generated in this video game. For real world applications we have the *ArcGIS CityEngine* that can assist in generating city layouts based on real world or imaginary data. Figure 2.1.3 shows a few examples of generated city layouts using the engine.

As mentioned PCG can be used to reduce file sizes and a prime example of this is the demo game *.kkrieger* which is only a measly $102KB$ in size, but it is able to generate a whole first person shooter game using procedural generation [13]. This is an extreme example of how procedural generation can be used to create a large amount of content with a very small file size. For reference the *jpeg* image in Figure 2.1.1 is a screenshot[1] from the game, and the file size of the image is $193KB$ which is about 1.9 times more data than the entire source code of the game.

This section will cover several methods of PG and discuss similarities, differences and most importantly limitations of the methods. Some of these methods have been used to experimentally generate music [15]. However we will try to

[1]This game was made for a competition and is a free demo, therefore no explicit rights are needed to include the picture here.


**Figure 2.1.1:** Screenshot from the game *.kkreiger*. All assets like textures and 3D models are procedually generated to save space.

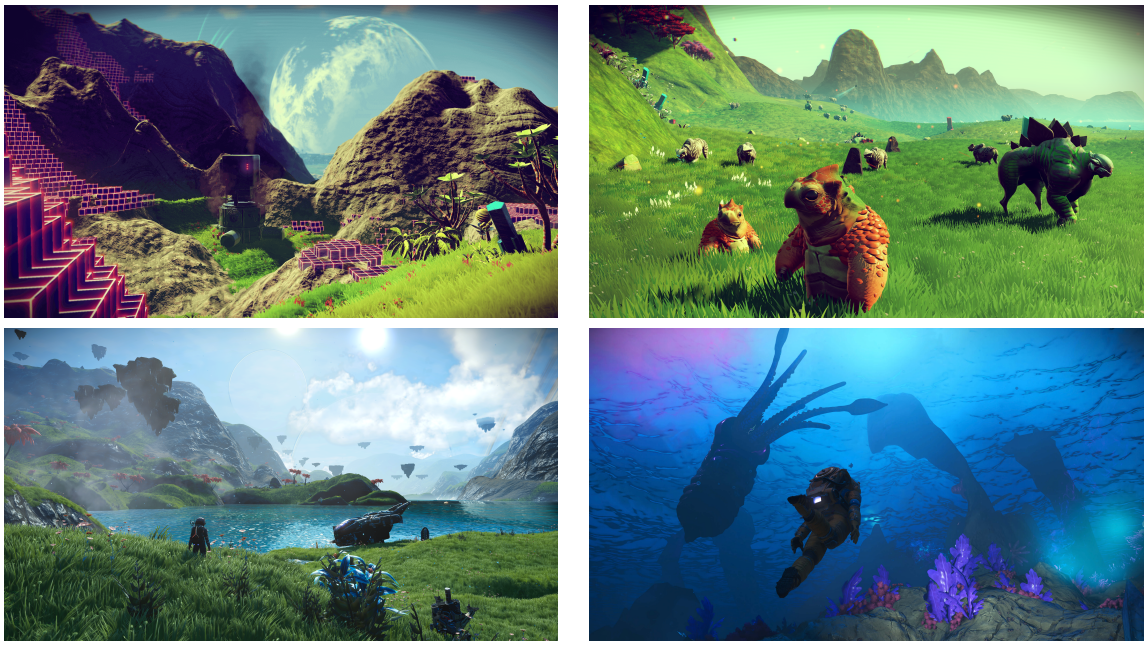


**Figure 2.1.2:** Example images from the game *No Man's Sky*. We can see both natural and "synthetic" landscapes as well as animals and plants generated by the game engine. The images are taken from the developers press kit [12].

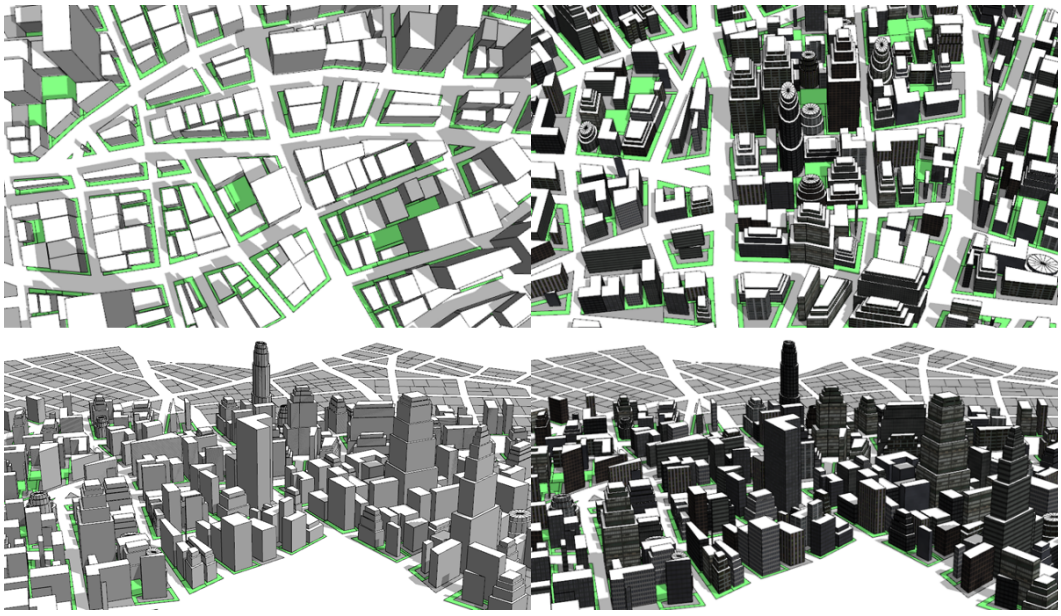


**Figure 2.1.3:** A few examples of city layouts generated using the *ArcGIS CityEngine* [14].

figure out which methods are suitable and maybe combine different methods to create something new.

### 2.1.1 L-systems (Lindenmayer systems)

Lindenmayer systems, usually shortened to L-systems, are rule-based rewriting systems originally introduced for modelling the growth of plants and other branching biological structures. An L-system starts from an initial string, often called the *axiom*, and repeatedly rewrites every symbol in that string according to a set of production rules. After each rewriting step the string becomes longer or more detailed, and the resulting sequence of symbols can then be interpreted geometrically, for example as line segments, turns, or branches. Because all symbols are rewritten in parallel, the method is well suited for modelling iterative growth.

In procedural generation this makes L-systems particularly useful when the goal is to create self-similar structure. A small set of rules can produce outputs that preserve an overall identity while still increasing in complexity from one iteration to the next. This is why the method is often associated with fractal geometry, botanical forms, and decorative branching patterns [3]. The strength of the method is that global visual complexity emerges from very compact local rules.

Figure 2.1.4 shows this clearly. The Sierpinski triangle demonstrates that L-systems can produce highly regular and mathematically rigid forms, while the plant and bush examples show that the same basic mechanism can also create more organic-looking structures. The arrowhead construction in Figure 2.1.5 further illustrates how repeated rewriting can gradually approximate a larger fractal shape through a sequence of simple local expansions.

A compact way to think about an L-system is that it begins from a short start-

ing string and repeatedly rewrites that string according to a small number of fixed rules. For the Sierpinski arrowhead curve, this means that a very small symbolic description is expanded step by step into a much longer drawing instruction. The important point is that the rewrite happens simultaneously for every symbol in the current string. In this way, a short rule set can expand into a detailed command sequence, and once that sequence is interpreted geometrically it produces the increasingly refined triangular structure seen in Figure 2.1.5. We can also look at a simple pattern where we start with the axiom[2]:

$$\texttt{A}$$

and the production rules:

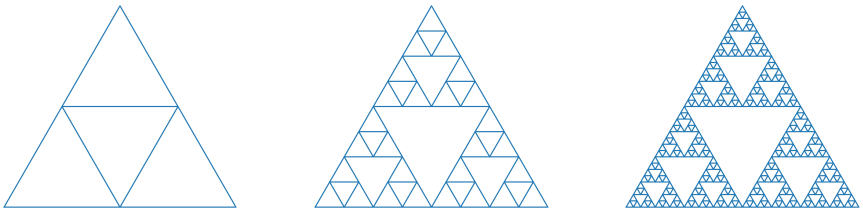
$$1 : \texttt{A} \rightarrow \texttt{AB}$$
$$2 : \texttt{B} \rightarrow \texttt{A}$$

These simple rules result in the following rather complex sequence after a few iterations:

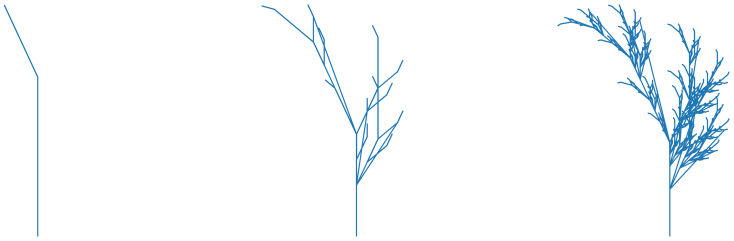
Axiom: A
$\rightarrow$ AB
$\rightarrow$ ABA
$\rightarrow$ ABAAB
$\rightarrow$ ABAABABA
$\rightarrow$ ABAABABAABAAB
$\rightarrow$ ABAABABAABAABABAABABA
$\rightarrow$ ABAABABAABAABABAABABAABABAABAABABAABAAB

The main limitation of L-systems is that they are strongest when the target structure can be described as iterative growth. They provide elegant control over repetition and branching, but they are less natural for domains where the important relationships are not hierarchical or do not arise from repeated rewriting. In practice, the quality of the output depends heavily on the chosen grammar and on how the generated symbols are interpreted geometrically.

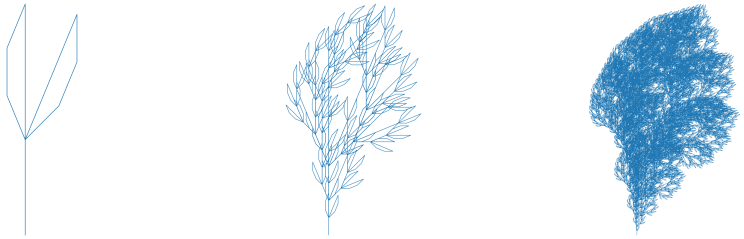
[2]the axiom can be thought of as the initial state of the system



(a) sierpinksi triangle

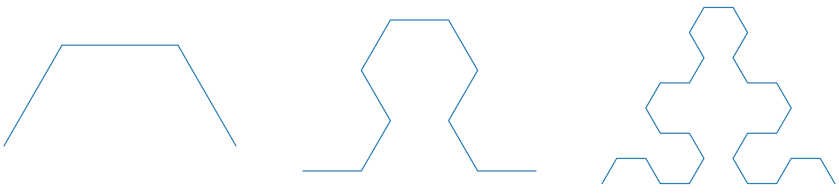


(b) fractal plant

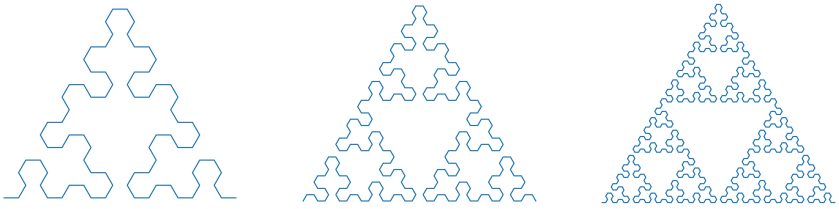


(c) fractal bush

**Figure 2.1.4:** Demonstration of the sierpinski triangle, a fractal plant and bush after one, three and five iterations repsectively.



(a) first iteration (b) second iteration (c) third iteration



(d) fourth iteration (e) fifth iterations (f) sixth iteration

**Figure 2.1.5:** Approximation of the sierpinski triangle using the sierpinski arrowhead curve.

### 2.1.2 Constraint satisfaction (CS)

Constraint Satisfaction (CS) approaches describe generation as a search problem. Instead of directly constructing an output step by step, the designer first specifies a set of variables, the domain of possible values for each variable, and a set of rules that describe which combinations are allowed [16]. The generator then searches for an assignment of values that satisfies all the constraints. In procedural generation this is attractive because it gives explicit control over the properties the output should have. The user is not only asking for randomness, but for randomness inside a carefully designed space of valid solutions.

In the following example, each cell in the map is a variable and each terrain type is one of the possible states in that variable's domain. The constraints describe which terrain types may be adjacent to one another and which global relationships should hold across the full map. Figure 2.1.6 shows the adjacency graph that defines the local rules, Figure 2.1.7 shows how the domain of each cell is reduced during propagation, and Figure 2.1.8 shows one final configuration satisfying the imposed rules. This is a simple example of how the method works, the algorthm can be a bit complex due to having to backtrack when a solution is not found, but the general idea is quite simple.

Constraint propagation is important because it reduces the search space before or during search. The Arc Consistency (AC)-3 algorithm is a standard example: it removes values that can no longer participate in any valid local assignment, thereby establishing arc consistency between neighbouring variables [17]. If a cell can only become *water* or *sand*, then the adjacent cells do not need to consider terrain types that are incompatible with both of those possibilities. This kind of pruning does not in itself produce a full solution, but it removes impossible choices early and makes the remaining search much more manageable.

A major strength of constraint-based procedural generation is therefore controllability. The designer can directly encode stylistic, structural, or logical requirements into the model. The main drawback is that this control comes at the cost of model design. Good constraint systems are often expressive and reliable, but they require the programmer to formulate the rules explicitly, and poorly chosen constraints can either leave the space too unconstrained or make it impossible to find any solution at all [16].

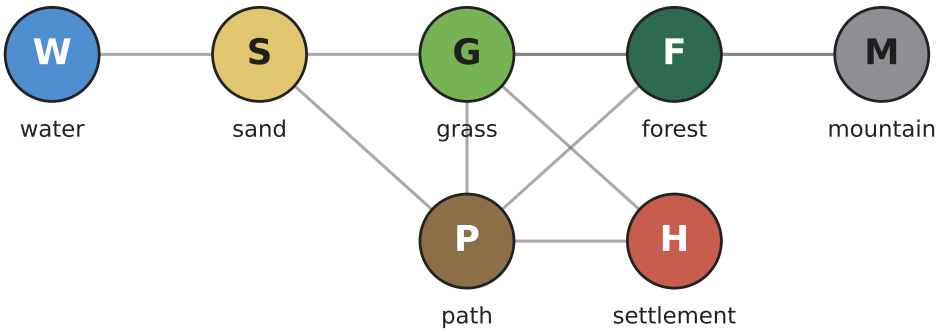



**Figure 2.1.6:** All different node types and how they are constrained. Each edge represents which note types can be adjacent in the domain.

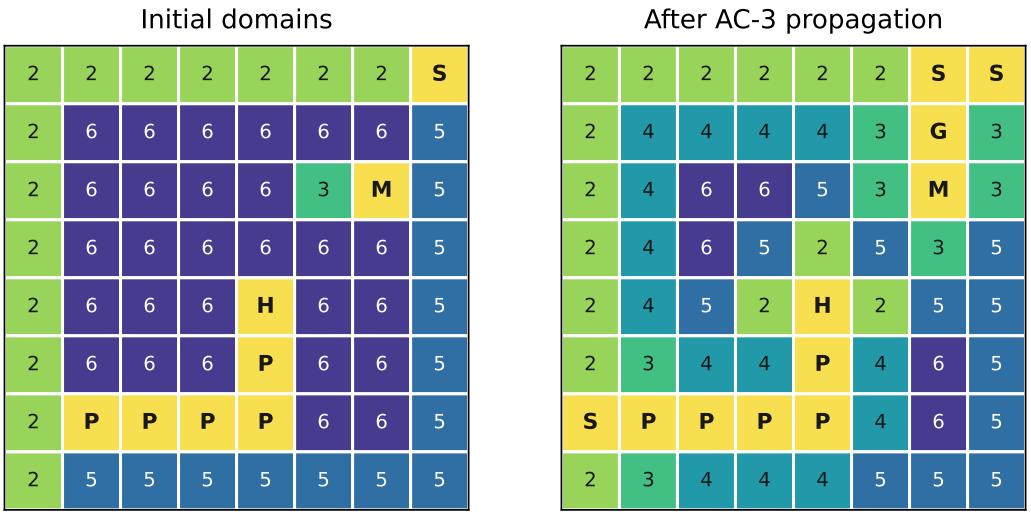



**Figure 2.1.7:** Initial domain and domain after AC-3 propagation. The AC-3 algorithm reduces the search space by ensuring arc consistency.

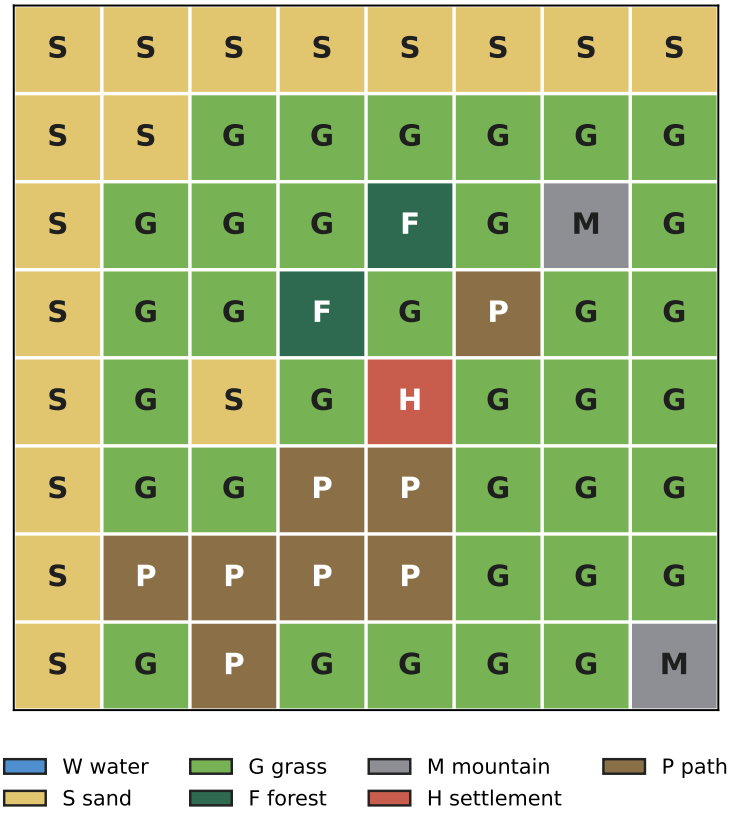



**Figure 2.1.8:** Final generated map satisfying all constraints.

### 2.1.3 Wave Function Collapse (WFC)

WFC can be understood as a constraint-based constructive generator that learns its local rules from an example rather than requiring all of them to be handwritten [18]. The input is typically a small exemplar image or tile arrangement. From this exemplar, the algorithm infers which local patterns are allowed to occur next to one another, and uses these learned adjacency rules to synthesize a larger output with a similar visual character. This makes WFC an interesting middle ground between explicit rule systems and example-based procedural generation [19].

The term *wave function collapse* is inspired by quantum mechanics, but here it is only a metaphor. The generated space begins in an undecided state where each cell can still take several possible values. As soon as one cell is observed, or *collapsed*, the compatible choices for neighbouring cells become more restricted. Propagation then removes choices that are no longer valid, and the process continues until the whole domain has been resolved. In other words, the method alternates between selecting a state and enforcing local consistency [18].

The rules for how the tiling should propagate are derived from an example input. In Figure 2.1.9 we see how the algorithm extracts local tile rules from a small exemplar and uses them as the basis for generating new content. These learned rules are what make the output consistently similar and ensure that it resembles the original example. This is the main difference between WFC and ordinary CS: in a pure constraint model the designer writes the rules explicitly, while in WFC the local rules are induced from the exemplar.

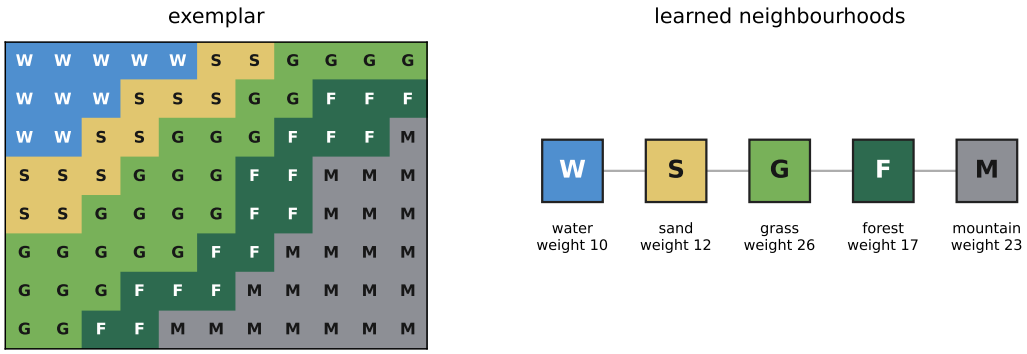



**Figure 2.1.9:** Given the exemplar on the left the WFC algorithm learns the local tile rules shown on the right.

In Figure 2.1.10 we can see how the number of possible states is reduced as soon as one state is collapsed. This reduction propagates outward, and the process of assigning states continues until the entire domain has collapsed.

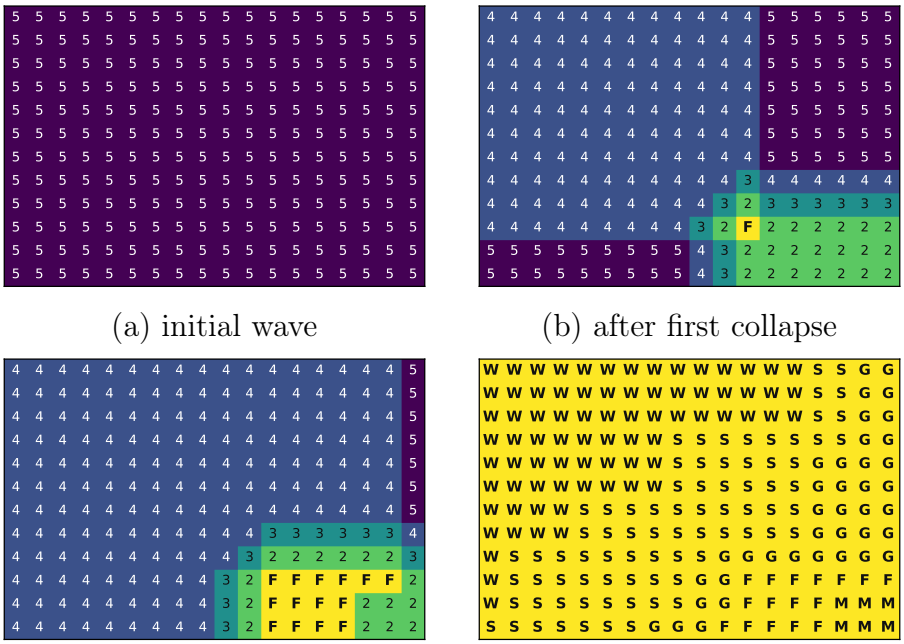



**Figure 2.1.10:** Shows how collapsing/observing a state propagates. An observation reduces the local uncertainty.

Once the rules for the tiling have been established, they can be used to generate new examples. In Figure 2.1.11 a larger domain is produced using the same learned rules as in the original exemplar.

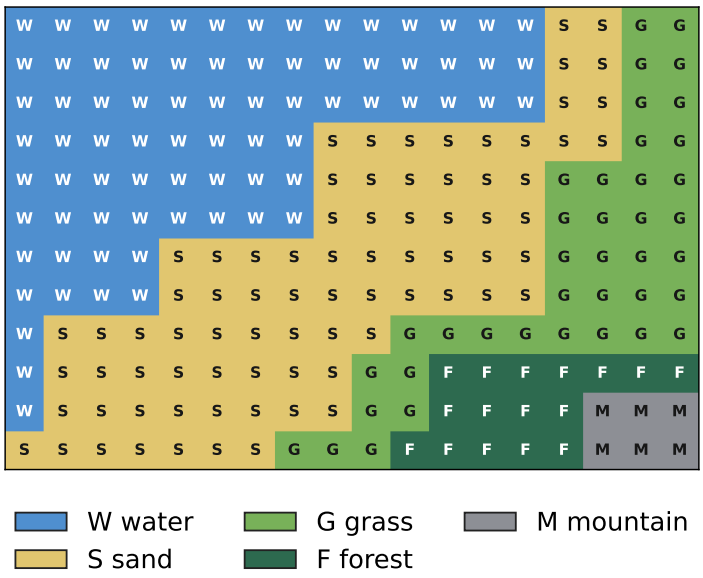



**Figure 2.1.11:** An example output for a larger domain than the original exemplar. The consistent rules lead to consistently similar outputs.

Finally, the learned rules do not imply that there is only one valid output. By changing the seed for generation, we can obtain visibly different results that

still adhere to the rules derived from the exemplar in the first step. This is an important strength of the method: it preserves stylistic consistency while still allowing variation.

The main limitation of WFC is that the model only reasons locally. It can be extremely effective at preserving texture, neighbourhood relationships, and low-level style, but larger structural goals are not guaranteed unless additional constraints or hierarchy are introduced. This becomes especially relevant when the target domain has strong long-range dependencies, such as architecture, level progression, or musical form [20].

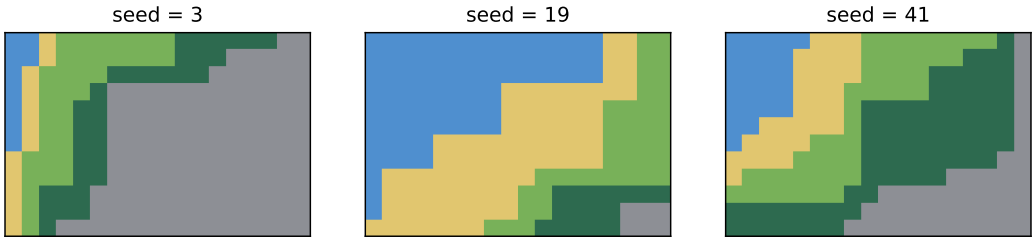



**Figure 2.1.12:** Illustration that the learned rules stay consistend while the seed changes the output.

### 2.1.4 Binary Space Partitioning (BSP)

Binary Space Partitioning (BSP) is a top-down method for dividing a space into smaller regions. In procedural generation it is especially common in dungeon and indoor level generation, where the objective is to create rooms, corridors, and a clear spatial hierarchy [21]. The basic idea is simple: start with one large domain, recursively split it into smaller partitions, and then place content inside those partitions. Because the regions are generated by subdivision, the resulting layout is inherently organized and non-overlapping.

Figures 2.1.13–2.1.15 illustrate this process. First, the domain is recursively divided into subregions. Then rooms are placed inside the partitions, and finally corridors are added to connect them. The last step rasterizes the abstract layout into a concrete dungeon-like map. This staged structure is one of the reasons BSP remains popular: the method separates *where* content may be placed from *what* content is actually placed there.

The main advantage of BSP is control over large-scale structure. Because the generator works hierarchically, it becomes relatively easy to influence room sizes, connectivity, and overall pacing of the environment. The drawback is that the results often inherit the bias of the partitioning scheme. Recursive splitting naturally favours rectilinear and architectural layouts, which is useful for dungeons and buildings, but less suitable for organic terrain or irregular natural structures [21].

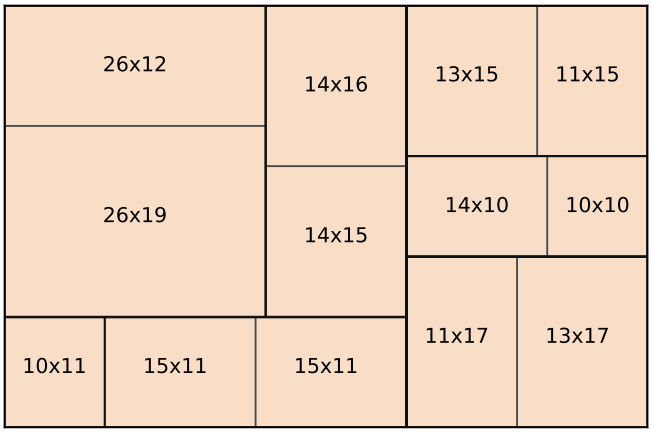



**Figure 2.1.13:** Initial domain partitioning, made by recursively divinging the domain in two.

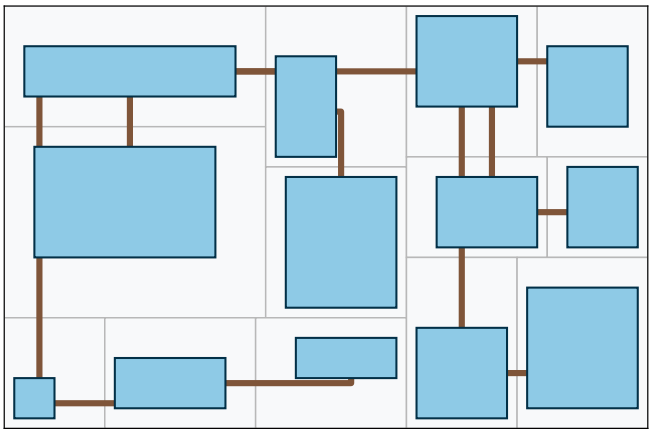


**Figure 2.1.14:** Generated rooms inside the partitions and connected them with corridors between rooms.

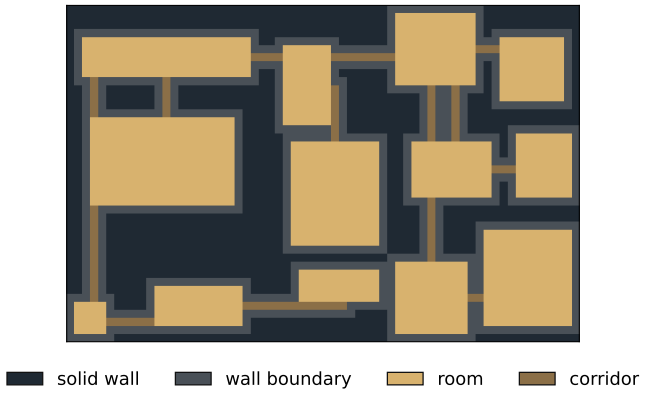



**Figure 2.1.15:** Rasterization of the generated dungeoun to show "final product".

### 2.1.5 Random noise and seeds

Noise functions are a common foundation for procedural generation because they provide continuous variation instead of isolated random values. A well-known example is Perlin noise, introduced as a smooth pseudo-random signal for image synthesis [22]. Unlike white noise, where neighbouring values are unrelated, gradient noise produces gradual transitions. Figure 2.1.16 shows a simple generated noise field of this kind. Such smooth variation makes noise useful for height maps, clouds, textures, and other domains where abrupt changes would look artificial.

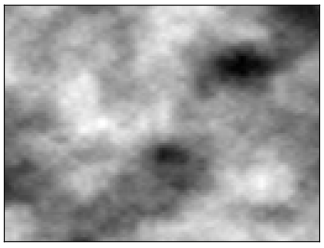


**Figure 2.1.16:** A simple example of smooth Perlin-like noise.

Another central idea is the use of a *seed*. The seed initializes the pseudo-random process and makes the output reproducible. If the seed and the parameters remain the same, the same generated result can be obtained again. Figure 2.1.17 shows this directly: changing the seed changes the concrete realization, while keeping the generation method fixed. This is extremely useful in procedural systems, since it allows variation without sacrificing reproducibility. Another benefit of using seeds it that it makes debugging and troubleshooting easier. If you get an error when running your code you can use the same seed to reproduce the error and figure out where it went wrong. This is not a feature of the seed itself but rather a very useful consequence of using seeds in procedural generation.

Noise also becomes more expressive when several frequencies are combined. Lower-frequency components define the broad shape of a terrain, while higher-frequency components add local detail. This layered construction, often described in terms of octaves or fractal Brownian motion, is shown in Figure 2.1.18. Once a smooth scalar field has been generated, it can be classified into discrete terrain types such as water, grass, rock, or mountain. In that way a continuous noise field becomes a terrain map or texture map rather than just an abstract signal [22, 3].

Figure 2.1.19 shows that the identity of the output is controlled not only by the seed but also by the parameters of the generator. By changing the scale, number of octaves, or relative contribution of different layers, we can make the same seeded process look smooth, rough, calm, or highly detailed. The strength of noise-based generation is therefore that it gives rich variation from a compact mathematical description. Its limitation is that raw noise does not by itself understand semantic structure: it can suggest terrain, but it does not know where a road, settlement, or melody should go without additional rules.

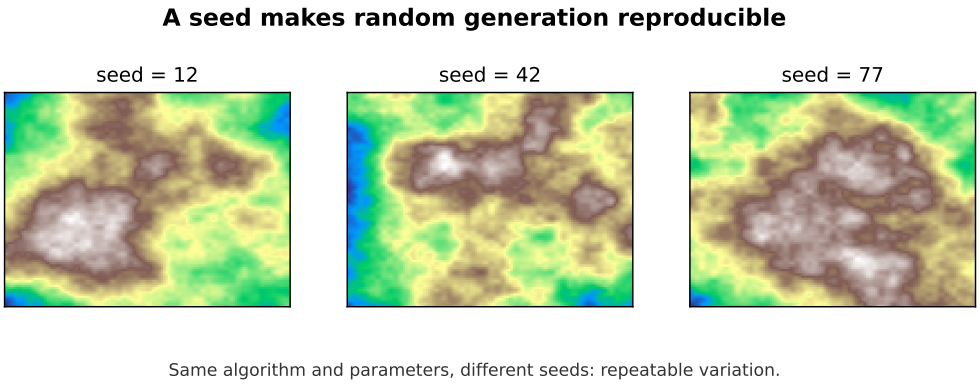



**Figure 2.1.17:** A seed makes random generation reproducible.

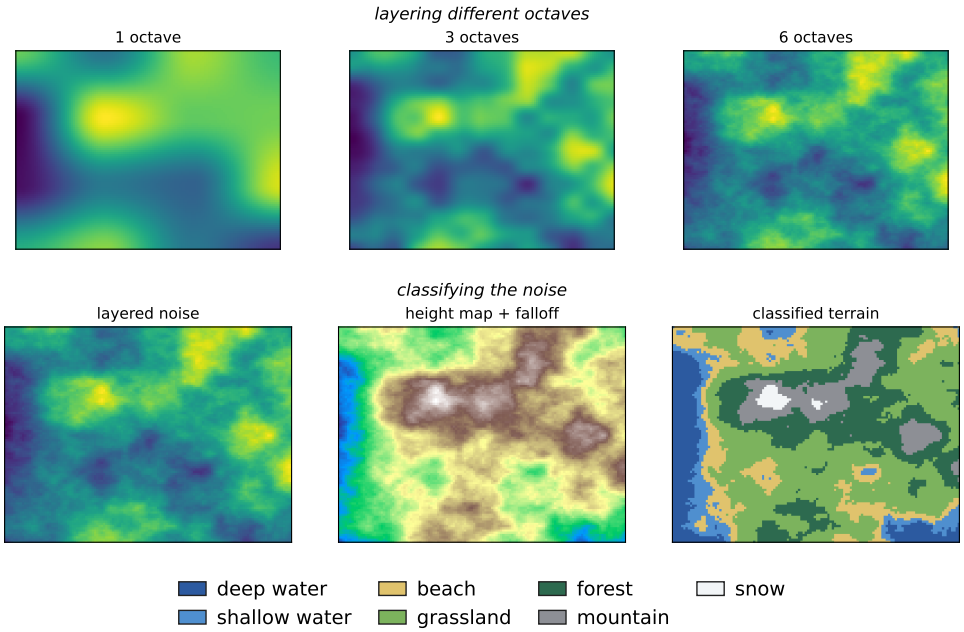



**Figure 2.1.18:** Using different octaves we can combine lower and higher frequency textures to create something more complex.

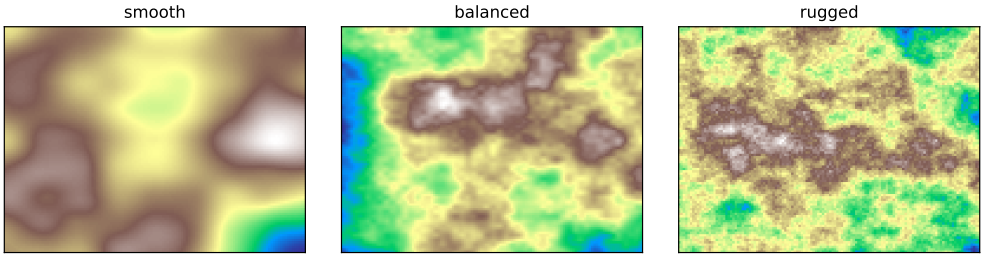



**Figure 2.1.19:** Using the same seed with different parameters we can control the identity of the output like style and scale.

### 2.1.6 Random walks

A random walk is a stochastic process in which an agent moves step by step according to local random choices [23]. In its simplest form each step is chosen independently of the previous history, which is why the classical example is often called the *drunkard's walk*. In procedural generation this makes random walks useful whenever the generated structure should emerge from local motion rather than from a global formula. Typical examples include cave carving, wandering paths, erosion-like traces, and density fields that can later be interpreted as textures or terrain.

The basic mechanism is simple: an entity moves through a domain, and its trajectory leaves some kind of mark. The path may directly become the output, for instance as a tunnel or a corridor, or it may accumulate density values that are interpreted afterwards. It is also common to limit each walker to a fixed number of steps and then start a new walk elsewhere. This creates a balance between local continuity and wider coverage of the domain.

In contrast to noise-based generation, random walks are strongly path dependent. The output is shaped not just by a value field but by where the walker has actually travelled. This makes the method especially suitable for organic structures with traces, streaks, or meandering boundaries. In Figure 2.1.20 the same underlying idea is used to generate three visually different textures. By changing the directional bias and the colouring of the accumulated walk density, the method can imitate wood grain, sand-like streaks, or an abstract terrain map. The wood example uses a predominantly vertical bias, the sand example a diagonal bias, while the map example uses a more even walk distribution that is later classified into colour bands.

The weakness of random-walk generation is that it can be difficult to control globally. Without additional bias, restarts, or post-processing, the output may become too sparse, too dense, or disconnected in undesirable ways. The method is therefore excellent for producing local organic variation, but less reliable when the target structure requires strong global organization [23].

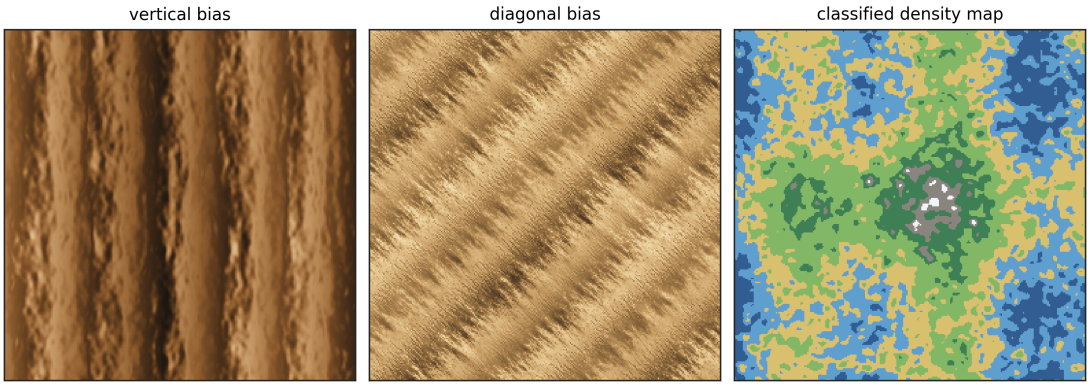



**Figure 2.1.20:** Three different textures generated using random walks.

### 2.1.7 Discussion

Now that we have looked at several methods of PG, some common themes become clear. All of the methods combine structure with variation, but they do so in different ways. L-systems create complexity through repeated rewriting, constraint satisfaction creates it through explicit validity rules, WFC learns local rules from examples, BSP organizes space through recursive subdivision, noise produces continuous variation through smooth pseudo-random fields, and random walks produce structure through accumulated local motion. In each case the generator is not merely choosing random values, but constraining randomness through a model. This means that these methods could be useful for music generation since they all involve some kind of constraint meaning if we can model the musical constraints within the bounds of the chosen method we would be able to generate coherent music.

Another useful distinction is the balance between local and global control. Some methods, such as random walks and basic noise fields, are excellent at producing natural-looking local variation but weak at guaranteeing larger structure. Others, such as BSP and constraint satisfaction, give stronger global organization at the cost of more explicit modelling effort. WFC sits somewhere in between: it preserves local style very effectively, but it does not automatically guarantee large-scale coherence. Which method is most appropriate therefore depends on whether the priority is texture, structure, control, or stylistic similarity.

This distinction is important for the thesis. The final goal is not simply to generate arbitrary variation, but to generate music that remains coherent under constraints. Methods such as noise and random walks are useful as background examples because they show how randomness can be shaped into recognizable outputs, but they do not by themselves provide the kind of rule-based control needed for melody and harmony. Constraint-based and example-driven methods are more relevant, since they suggest ways of combining variation with explicit structural requirements [16, 18, 20]. The following chapters build on that idea and investigate how a procedural music generator can gradually move from simple variation toward more controlled musical structure.

No method has been choosen yet as a result of looking at the different methods, however through the iteration process we will try to figure out what method feels natural and copatible with the music theory we will use as the ground rules for the generator.

## 2.2 Music theory

This section contains some basic music theory relevant to understand how music is constructed. It is not neccessary to understand everything in depth, but farmiliarity with these concepts will be of great help to understand the musical aspect of the thesis. Throughout the thesis there will be several examples and figures using musical notation so a basic understanding of how to read these figures will be benefitial.

### 2.2.1 Notes

The very core of music is composed of notes. Notes are vibration in air which we interpret as pitches. In the western system we agree upon the standard of pitches is based around $A_4$ which is $440Hz$. All other pitches are tuned in relation to this reference.

The musical note is represented on the staff by a circular symbol called the note-head. The vertical placement of the note-head on the staff determines which pitch it represents. This is illustrated in Figure 2.2.1 where we can see the placement of the notes from $C_2$ up to $C_5$. Notice how only the white keys form the piano are notated. The black keys are notated with the ♯- and ♭-symbols. The ♯-symbol is pronounced "sharp" and raises the note by a semitone and the ♭-symbol is pronounced flat and lowers the note by a semitone. The horizontal placement is not directly an indication of time or duration, the temporal aspect is rather defined symbolically. The note can also have a stem, which is a vertical line attached to the note-head. The direction of the stem (up or down) does not change the pitch, but it can affect the visual layout of the music. When multiple notes are close together, stems can be directed in opposite directions to make the differentiation of voices clearer.

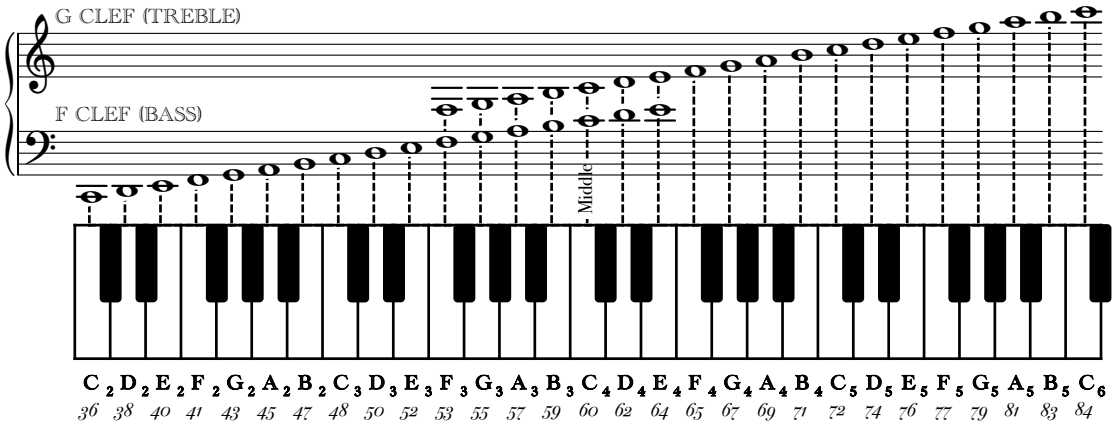



**Figure 2.2.1:** Overview of note placement on the staff and piano, additionally the corresponding midi numbers are written below the note names. Notice how the number increases by two when there is a black key between consecutive notes.

The same musical pitch can be written in several different ways in order to dictate the duration of the note. This is done by using variations of the note. This can be by refering to Figure 2.2.2 we can see the different parts of the note

highlighted with colour. The note-head is black and the placement of this decides which musical pitch is represented. The stem can go either up or down, this does not change the meaning it just makes it more compact to make it go down if the note placement is high on the staff. The direction of the stem can carry meaning if there are two notes simultaneously on a staff, then it usually indicates two different voices. Meaning one player playes the melody with stem up and one with the stem down. Next we have the beams and the flags coloured in purple and blue respectively. These two carry the same meaning, one flag is the same as one beam, meaning they have the same duration. The beam simply makes it less visually crowded when reading. The omitting of beams and only using flags *can* mean that the notes are more separated, but this is not neccessarily the case.

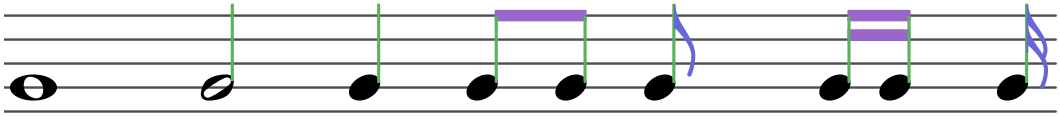


**Figure 2.2.2:** The figure shows the parts of the musical note. The note-head in red, stem in green, beam in purple and flag in blue.

**Table 2.2.1:** Table with common note values, their names and their corresponding duration.

| Symbol | Name | Rest symbol | Duration |
|---|---|---|---|
| 𝅝 | Whole note | 𝄻 | 4 beats |
| 𝅗𝅥 | Half note | 𝄼 | 2 beats |
| ♩ | Quarter note | 𝄽 | 1 beat |
| ♪ | Eigth note | 𝄾 | 1/2 beat |
| 𝅘𝅥𝅯 | Sixthteenth note | 𝄿 | 1/4 beat |
| 𝅘𝅥𝅰 | Thirtysecond note | 𝅀 | 1/8 beat |

## 2.2.2 Scales and Keys

The musical scale is building block for creating not only melodies but also harmony, which we will cover a bit later. Firstly the musical scale is an orderet set of musical notes. In the western canon this is usually seven notes. You can see the basic c-major scale in Figure 2.2.1, it is comprised of the notes C D E F G A B The tones of the scale can be named according to their function within the scale. We often label the notes $\hat{1}$, $\hat{2}$, $\hat{3}$, $\hat{4}$, $\hat{5}$, $\hat{6}$, $\hat{7}$, and then starting back at $\hat{1}$. The use of the carets are called the Nashville Number System, this is used to differentiate the scale degrees from the chord tones which will be discussed in Section 2.2.4. We can also alter these degree numbers using the ♭- and ♯-symbols for instance the minor scale would be notated $\hat{1}$, $\hat{2}$, $\flat\hat{3}$, $\hat{4}$, $\hat{5}$, $\flat\hat{6}$, $\flat\hat{7}$. And the notes in the C-minor

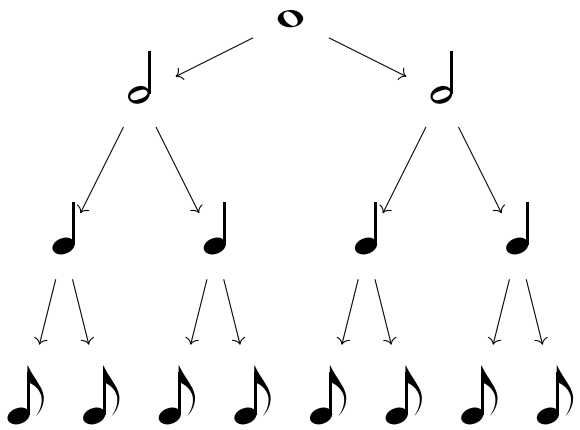


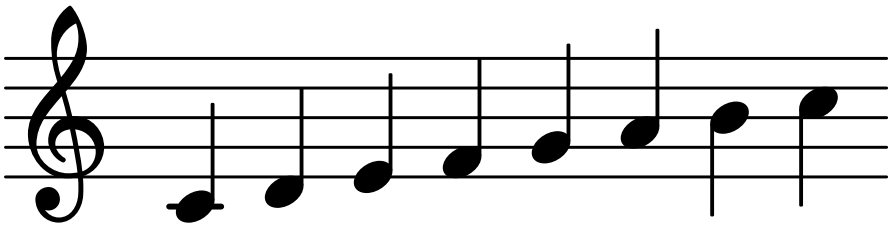
**Figure 2.2.3:** Tree sctructure showing how you can divide the whole note into the smaller note values. Here each level in the tree takes the same amount of time, meaning the whole note at the top takes as long as the eight eigth-notes at the bottom.

scale would be C D E♭ F G A♭ B♭. The placement of the sharp- and flat-symbols before or after comes from the pronunciation of the notes, we say "flat three" and "sharp four", but we say "E flat" and "F sharp". When we consider chord structures we use the numbers but usually count above from 8 to signify harmonic functions, but when considering only scales and melodies we usually only use 1 through 7.

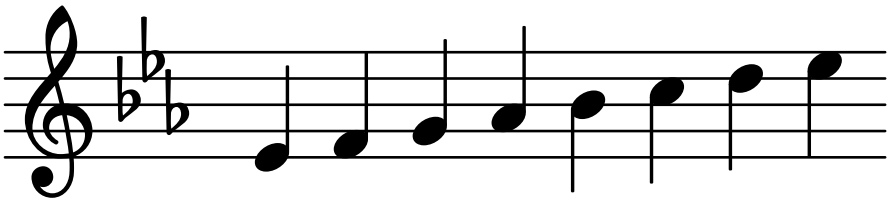
The different notes in the scale have different functional names, these names are not tied to the pitch, meaning whether it is a C or a E♭ the functional name of the note is the same as long as it is the same scale degree. The most important function is that of the scale degree $\hat{1}$, this is the *tonic*, which is the 'home' of the key. This is the most stable function there is, it feels resolved and does not have any tension urging it to move anywhere and feels very stable. In Table 2.2.2 you can see the common names for the scale degrees. These fancy names simply give some context to what functions the notes usually play in a melody. The leading tone has a lot of tension and wants to resolve to the tonic above.

**Table 2.2.2:** Scale degrees and their funcitonal names.

| **Scale-Degree Number** | **Scale-Degree Name** |
|---|---|
| 1 | Tonic |
| 2 | Supertonic |
| 3 | Mediant |
| 4 | Subdominant |
| 5 | Dominant |
| 6 | Submediant |
| 7 | Leading tone |



**Example 2.2.1:** The C-major scale written in the treble clef. Listen in the online archive.



**Example 2.2.2:** The E♭-major scale written in the treble clef. Listen in the online archive.

The concept of a musical key is simply shifting around where the tonic note is. For instance in Example 2.2.1 you can see the major scale in C and Example 2.2.2 shows the major scale in E♭. These two scales sound "the same" meaning the relationship betweenall the notes are the same but the actual pitches are different. The key is simply the name of the tonic note, so the first example is in the key of C-major and the second example is in the key of E♭-major. The concept of a key is important because it gives us a reference point for understanding the function of notes and chords within a piece of music. It also helps us to understand how melodies and harmonies are constructed and how they relate to each other.

### 2.2.3 Intervals

Before discussing harmony we need to consider the different types of distances between notes. The musical distance between two notes are called an *interval*. Intervals can be identified by the number of semitones between the notes. All the intervals the intervals have special names and some distances can have different names depending on how they are spelled[3], these names can be seen in Table 2.2.3. The list also provides the number of semitones between the notes.

In Figure 2.2.4 we can see how all the normal intervals look on a stave. Naming the interval starts by finding the "general distance", meaning we look at the note name. If we consider the interval between C and E♭ we first consider that from C to E we have three letters. C, D then E, which means we have a third, then we look at the variant. Since the E is flattened to E♭ we have the minor third since the distance is 3 semitones instead of 4. You can see this in Example 2.2.3, here we observe that the distance between the notes in a minor third and a major third

[3]the same pitch can be "spelled" in different ways, for instance A♭ and G♯ are written differently, but sound the same since they are the same pitch

**Table 2.2.3:** Table containing all musical intervals up to an octave, the number of semitones and their abbreviated form.

| Interval name | Semitones | Abbreviation |
|---|---|---|
| Unison | 0 | P1 |
| Minor second | 1 | P1 |
| Major second | 2 | m2 |
| Minor third | 3 | m3 |
| Major third | 4 | M3 |
| Perfect fourth | 5 | P4 |
| Tritone[4] | 6 | TT or aug4/dim5 |
| Perfect fifth | 7 | P5 |
| Minor sixth | 8 | m6 |
| Major sixth | 9 | M6 |
| Minor seventh | 10 | m7 |
| Major seventh | 11 | M7 |
| Octave | 12 | P8 |

is the same, the only difference is that the minor third is one semitone smaller, but that is shown by the ♭-symbol which means the note has been lowered by a semitone. Consequently for the tritone we see there are two ways to spell the same interval, one is a fourth that has been raised and the other is a fifth that has been lowered. These two are *enharmonically equivilant* which just means that they are the same pitches, they are just spelled differently. For the augmented fourth(the first spelling of the tritone) we se the ♯-symbol used, which simply means the note is raised by a semitone.

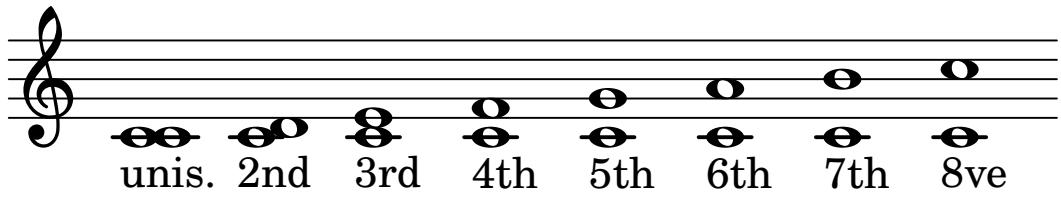
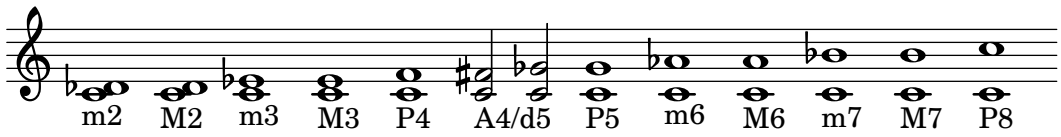



**Figure 2.2.4:** All normal intervals from the note $C_4$ devoid their variants in quality.

### 2.2.4 Harmony

Harmony is the vertical component of the music, harmony is what happens when two or more notes happen at the same time. This is what chords are build around. When considering the harmonic progression in tonal music we can consider all the standard chords in the major key. They are labeled by numbers using the roman numerals. Upper case means the chord in major and lower case means minor.

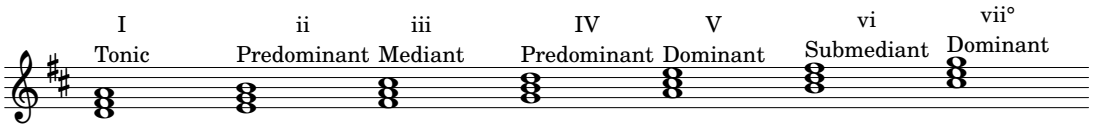
[4]can also be called diminished fifth or augmented fourth, where diminished simply means lowered by semitone and augmented means raised by one semitone




**Example 2.2.3:** All the musical intervals from the unison to the octave including their variants. Listen in the online archive.

You can see the difference between major and minor chords in Example 2.2.5, and listen to the difference at the link provided in the caption.

Example 2.2.4 shows the full set of diatonic triads in D major. The Roman numerals make the chord quality clear, and the functional labels connect each triad to its role in the key. This is the harmonic material that forms the basis for the progression patterns discussed below.




**Example 2.2.4:** The diatonic triads in D major with Roman numerals and functional names. Listen in the online archive.

Looking at Figure 2.2.5 we can see the typical flow of harmonic functions in tonal music. Tonic means the home, the most stable function. Dominant is the function with the most tension. Pre-dominant usually leads into the dominant or sets the expectation for the dominant. The mediant and submediant are usually not considered to be part of the main harmonic progression, but they can be used to create interesting harmonic progressions and add some color to the music. The arrows indicate the typical flow of harmonic functions, however this is not a strict rule.

In harmonic context we often talk about cadences. Cadences are the classifications of the motion between chords. The most common cadence is the Perfect Authentic Cadence (PAC), this is when we move from the dominant function to the tonic function, for instance from V to I. This is the most stable and resolved cadence there is, it is the musical equivalent of a full stop in language. The next most common cadence is the Imperfect Authentic Cadence (IAC), this is when we move from the dominant function to any other function than the tonic, for instance from V to vi. This is a less resolved cadence and it can be used to create tension and lead us somewhere else than home. The last common cadence is the Half Cadence (HC), this is when we move from any function to the dominant function, for instance from IV to V. This creates tension and leaves us hanging on the dominant function which usually leads us back to the tonic. There are many other types of cadences, but these are the most common ones. [24]

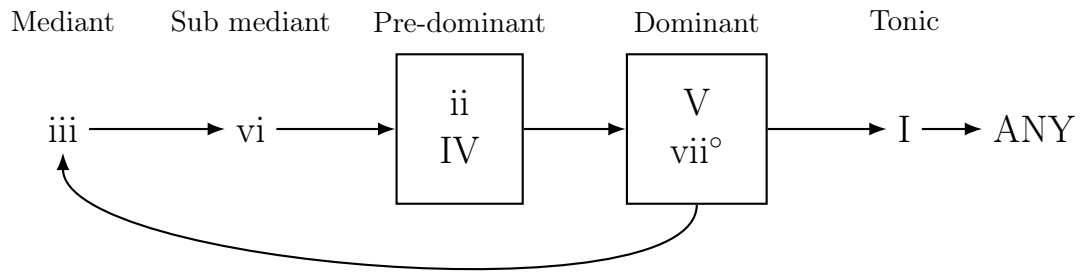
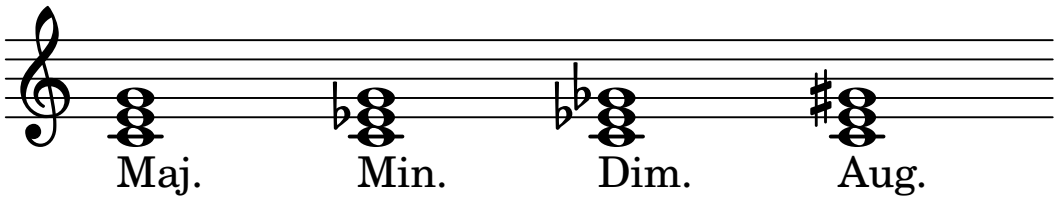



**Figure 2.2.5:** Harmonic progression in tonal music. The arrows indicate the typical flow of harmonic functions.




**Example 2.2.5:** The four main triads(chords with three notes), major, minor, diminished and augmented. These are the most basic chords, but major and minor are the most common ones. Listen in the online archive.

### 2.2.5 Time signatures

The time signature explains what is in a bar of music. A bar is the space between the vertical lines on the staff. A 4/4, sometimes written as **C**, time means that there are four quarter notes per bar and consequently 3/4 time means that there are three quarter notes per bar. For this thesis I will only employ the standard 4/4 time signature, it is the most common and straightforward time signature.

### 2.2.6 Musical form

Now that we have an idea of the musical structures on the melodic and harmonic layer we will take a look at musical forms. This is a very important idea when it comes to constructing melodies. A melody is not an infinitely wandering idea of music. A good melody follows a form. Often this can be viewed as repetitions and variations of a given musical idea. Two very common forms of musical form are the *sentence form* and the *period form* [25, pp.20-24].

In Example 2.2.6 we can see an example of the musical sentence. It starts

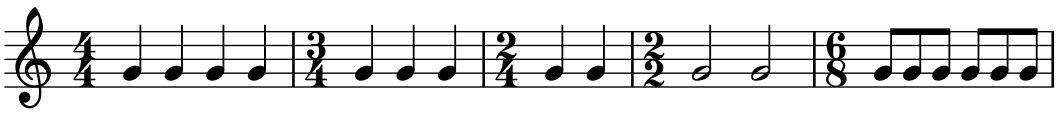
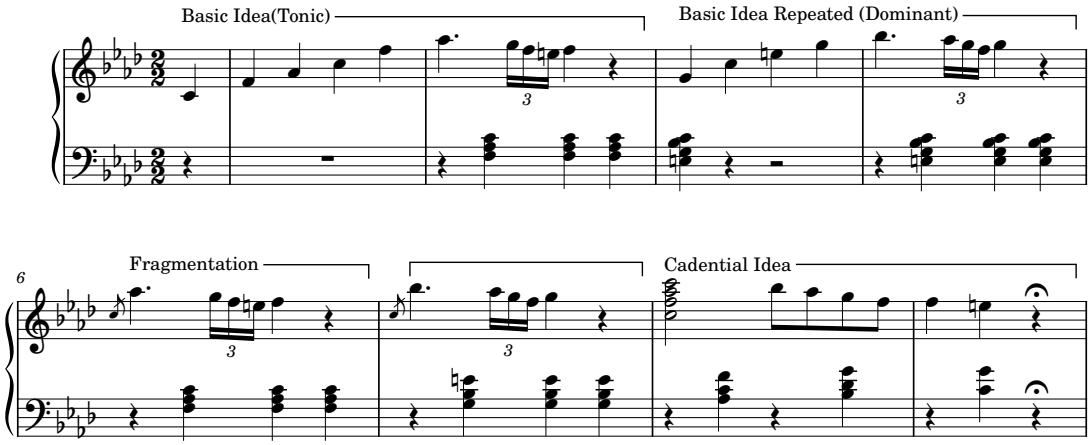


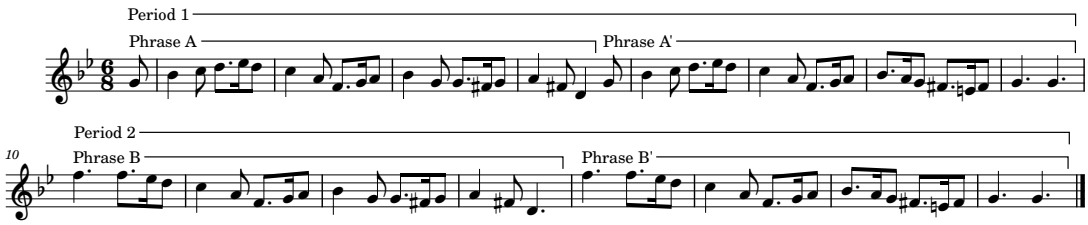
**Figure 2.2.6:** Demonstration of a few common time signatures and how the type of note is the lower number and the amount is the higher. For instance, $\frac{3}{4}$ has *three* instances of the *quarter* note.

with a 2-bar opnening phrase, labeled as the *basic idea*, then we have a repetition of the same idea only adapted for a new chord. Then the following 2 bars are a fragmentation of the idea. Then for the last two bars we have a conclusion, this ties the idea together. The cadence at the end can be considered the end of a sentence. It concludes the idea before it moves on to something else.




**Example 2.2.6:** A classical example of the musical sentence form. These are the opening bars of Beethovens piano sonata in F-minor [26]. Listen in the online archive

In Example 2.2.7 we see an example of the musical period. This example is from the British folk song *Greensleves*. The period is a form that is very similar to the sentence, but it has a more balanced structure. It consists of two phrases, the first one is called the *antecedent* and the second one is called the *consequent*. The antecedent phrase ends with a half cadence, which creates tension and leads us to the consequent phrase which ends with a perfect authentic cadence, which resolves the tension and gives us a sense of closure.




**Example 2.2.7:** A classical example of the musical period form. This is the melody of the British folk song *Greensleves* [27]. Listen in the online archive

## 2.3 Object oriented programming

OOP is a way of structuring software around *objects*. An object combines data and behaviour, meaning that it stores information and also provides operations that can act on that information. Instead of treating a program as one long sequence of instructions, the program is divided into units that represent meaningful concepts in the problem domain [28, 29].

In practical terms this usually means that the programmer defines a *class*, which acts as a template, and then creates *objects* from that class. Figure 2.3.1 illustrates this idea using a simple melody example. The class describes what information a melody should contain and what operations are available, while the object is one concrete melody with specific values.

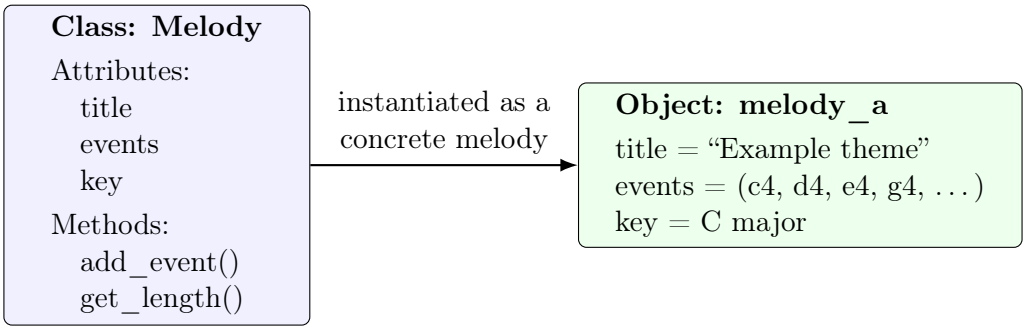



**Figure 2.3.1:** A simple illustration of the difference between a class and an object. A class describes what kind of data and behaviour an object has, while an object stores one concrete instance.

### 2.3.1 Encapsulation and abstraction

Two central ideas in OOP are encapsulation and abstraction. Encapsulation means that related data and behaviour are grouped together. For example, instead of storing note events in one place and melody-related operations in another, they can be kept together in a single object. This makes the program easier to reason about because the responsibilities of each object become clearer.

Abstraction means that a class can expose the parts of the system that are important for the user of the class, while hiding unnecessary implementation details. In a well-structured program this reduces complexity, because a developer can interact with a class through a small set of meaningful operations instead of having to understand every internal detail immediately.

### 2.3.2 Composition and inheritance

Another important idea in OOP is the relationship between objects. One common relationship is *composition*, where one object consists of one or more other objects. A simple example is that a melody can consist of several note objects. This is often a natural way to model structured data.

A second common relationship is *inheritance*, where a class derives behaviour from a more general parent class. Inheritance can be useful when several classes share a lot of common structure. However, inheritance is not always the best

solution. In many cases it is simpler and clearer to combine several smaller objects instead of building a deep hierarchy of subclasses.

### 2.3.3 Polymorphism

Polymorphism means that several different classes can be used through the same interface. This allows a program to treat them uniformly even though their internal behaviour differs. Figure 2.3.2 shows a simple example of this idea. Different constraint classes can implement the same operation, for instance checking whether a musical candidate satisfies a certain rule, while each class applies a different criterion.

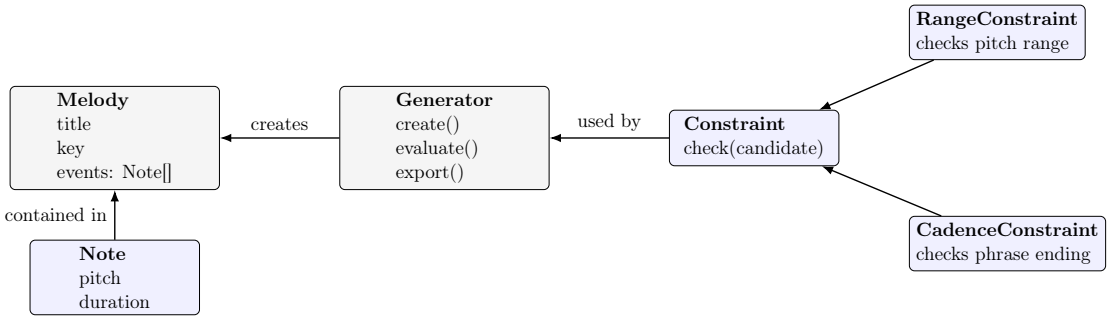



**Figure 2.3.2:** A simple object-oriented example showing two important ideas. Composition is used when a melody consists of note objects, while polymorphism is used when different constraint classes can be treated through one shared interface.

This is useful because it allows a system to be extended without rewriting the code that uses the interface. New rules or behaviours can be introduced by defining a new class that follows the same pattern as the existing ones.

### 2.3.4 Relevance to the thesis

In this thesis OOP is relevant because the project works with structured musical concepts rather than only raw numbers. Notes, melodies, motifs, harmonic structures, and constraints are easier to reason about when they are represented as explicit objects. This does not mean that OOP alone solves the problem of generating good music, but it provides a useful way of organising the software such that the program structure can reflect the musical structure being discussed.

For this reason OOP is part of the theoretical background of the project. Later in the thesis, when the actual iterations and software architecture are discussed, these ideas become relevant in a more concrete way. The purpose of this section is therefore only to introduce the central concepts so that the design decisions in the later chapters are easier to follow.

### 2.3.5 Strengths and limitations

The main strength of OOP is that it can make a large program easier to structure, understand, and extend. By giving different responsibilities to different classes, the code can be decomposed into smaller and more meaningful parts. This is especially useful when the problem domain itself has many distinguishable entities.

At the same time, OOP is not automatically a good design. Poorly chosen classes can make the program more confusing rather than less. Too much abstraction can make the code difficult to inspect, while unnecessary inheritance can create rigid structures that are hard to modify. Therefore the value of OOP depends on whether the object model matches the actual problem being solved.

## 2.4 Iterative development

> A major problem in the design of a computer system is that many aspects of system behaviour and performance are not discovered until the system has been built...[30]
>
> *F.W. Zurcher*
> *B. Randell*

Iterative development is a way of building a system through repeated cycles of implementation, evaluation, and revision. Instead of trying to design the entire system perfectly from the beginning, a smaller version is built first, then examined to identify weaknesses, missing functionality, or new opportunities. The next version is then developed using the knowledge gained from the previous one.

This approach is especially useful when the problem is only partly understood at the beginning, or when the behaviour of the system cannot be fully predicted from planning alone. In such cases, implementation itself becomes a way of learning about the problem.

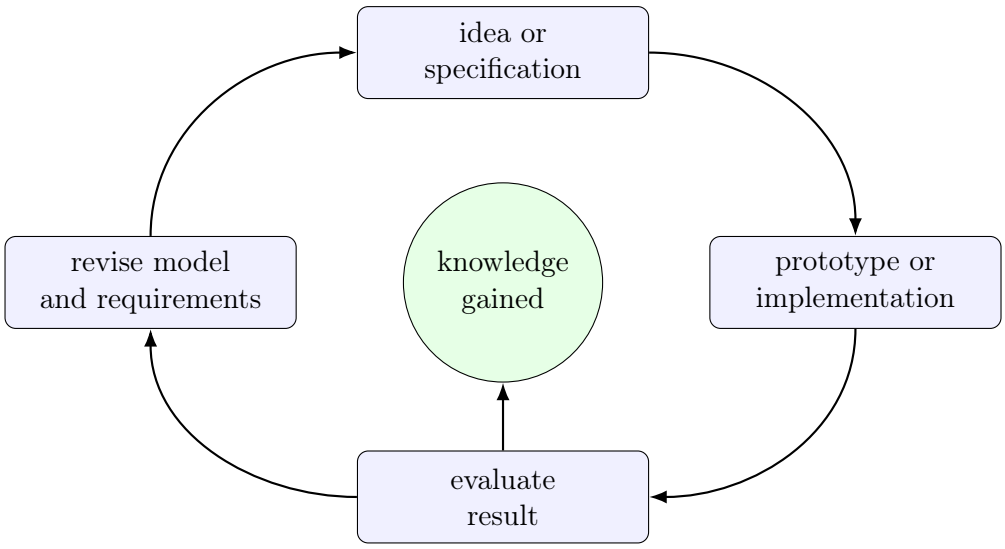



**Figure 2.4.1:** A simplified iterative development cycle. Each cycle begins with an idea, produces a prototype, evaluates the result, and revises the design before the next cycle begins. The knowledge gained from each stage informs the overall process.

### 2.4.1 Prototype and evaluation

A central part of iterative development is the use of prototypes. A prototype does not need to solve the whole problem well, but it should be useful for exposing important properties of the system. By building something early, it becomes possible to observe practical issues that may not be visible in an abstract design.

This means that evaluation is not only something that happens at the end of a project. Instead, evaluation becomes part of the development method itself. After each version, the developer can ask what works, what fails, what is missing, and which assumptions need to be changed before the next version is attempted.

### 2.4.2 Why iterative development is useful

The main strength of iterative development is that it allows the design to improve in response to concrete observations. This can reduce the risk of spending a large amount of time refining an initial plan that turns out to be based on poor assumptions. It also makes the process more adaptable, since new requirements or insights can be incorporated between iterations instead of forcing them into an unsuitable original structure.

Another strength is that the process naturally produces intermediate results. Even if early versions are limited, they often reveal which parts of the system are easiest to solve, which parts are hardest, and which abstractions are most useful. This knowledge can then guide the next design decisions.

### 2.4.3 Limitations

Iterative development is not without drawbacks. Repeated redesign can lead to extra work, especially if an early version is built on assumptions that later prove to be incompatible with the desired final system. There is also a risk that the project becomes a sequence of local fixes rather than a coherent whole if the lessons from one iteration are not properly summarized and carried forward.

For this reason an iterative process still requires reflection and planning. The value of the method does not come from rewriting the system repeatedly for its own sake, but from using each revision to make the next version better informed than the last.

### 2.4.4 Relevance to the thesis

Iterative development is relevant to this thesis because the project concerns a creative and technically uncertain problem. A melody generator is not only a software system, but also a musical one, meaning that its quality cannot be judged solely by whether it runs correctly. It must also be evaluated according to how coherent, singable, and useful the generated musical output is.

Because of this, a purely top-down design would have been difficult. Many of the important requirements become visible only after hearing and inspecting generated examples. An iterative process is therefore a suitable background method for the project: it supports prototyping, evaluation of musical output, and revision of both the code structure and the musical rules between versions.

## 2.5 AI in software development

Artificial intelligence is rapidly becoming part of everyday software engineering practice. What only a few years ago looked experimental is now increasingly presented as ordinary development tooling. Coding assistants, chat-based code generation, automated review systems, and agent-based tools are no longer discussed only as research prototypes, but as products that are actively being integrated into professional workflows [31, 32, 33]. For this reason, the question for this thesis is not simply whether AI can be used in software development, but how it should be used responsibly and effectively.

### 2.5.1 Increasing adoption

Several recent surveys suggest that AI tools are already widespread among developers. While the exact percentages vary depending on the survey population and the wording of the question, the overall trend is consistent: developers are using these tools, organizations are investing in them, and their role in the development workflow is growing [31, 32, 33]. Figure 2.5.1 summarizes a few representative results from widely cited industry reports. The exact numbers should not be treated as directly comparable, since the surveys ask slightly different questions, but together they illustrate that AI-assisted development has moved well beyond early experimentation.

In practical terms, this means that AI is increasingly becoming part of the normal tooling landscape of software engineering, much like version control, testing frameworks, or static analysis tools. It is therefore reasonable for a thesis about software development to treat AI not as a novelty, but as a serious development aid that deserves explicit reflection [32, 33].

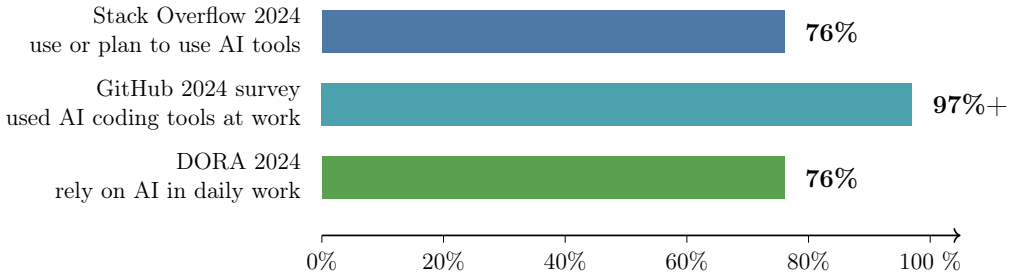



**Figure 2.5.1:** Selected survey results illustrating the prevalence of AI-assisted software development. The figures are based on different populations and question formulations, so they should be read as trend indicators rather than directly comparable measurements.

### 2.5.2 Potential benefits

One important reason for the rapid adoption of AI tools is that they appear to offer real gains in productivity and workflow support. These gains do not necessarily mean that the model replaces the developer. Rather, the benefit often lies in assisting with time-consuming or repetitive tasks such as code completion, boilerplate generation, refactoring suggestions, test drafting, documentation support,

and codebase exploration. This can allow the developer to spend a greater proportion of time on design decisions, debugging strategy, and evaluation of alternative solutions.

Recent empirical work also suggests that these benefits are not only anecdotal. Controlled and field-based studies indicate that access to coding assistants can increase the rate at which software tasks are completed, although the size of the effect depends on context, task type, and developer experience [34]. At the same time, many studies and reports frame the benefit less as fully autonomous programming and more as augmented development, where the developer remains responsible for understanding and validating the result [35, 36, 37].

### 2.5.3 Limitations

At the same time, increased adoption does not imply unconditional trust. One of the clearest themes in recent literature and survey material is that developers often find AI useful while still being skeptical of its output [31, 38]. This skepticism is well justified. AI-generated code may be syntactically correct while still misunderstanding requirements, introducing subtle bugs, using poor abstractions, or suggesting insecure solutions. It can also create the false impression that a difficult problem has been solved simply because a fluent-looking answer has been produced.

For this reason, the value of AI in software development depends heavily on verification, domain understanding, and responsible use. A developer who relies on AI without understanding the surrounding system risks turning the tool into a crutch. By contrast, a developer who already understands the problem domain can use AI more effectively as an assistant: to accelerate drafting, explore alternatives, and reduce routine effort while still retaining control of the technical decisions [38].

This perspective aligns well with the idea of appropriate trust. The goal is not to reject AI, but to use it with enough understanding that its contributions can be evaluated critically. In other words, AI can improve development speed and convenience, but only if the human developer remains responsible for interpretation, validation, and final judgment [31, 38, 33].

### 2.5.4 Relevance to this thesis

In this thesis, AI is not treated as a replacement for software engineering work, but as a tool that may support the development process. This is the reason the first two iterations were developed without AI assistance. Doing the early work manually helped establish a basic understanding of the musical domain, the program structure, and the design trade-offs involved in the project. Without that foundation, it would be much harder to judge whether AI-generated suggestions were correct, useful, or aligned with the goals of the system.

The third iteration therefore uses AI in a more deliberate way. By that stage, the project has already accumulated enough structure and domain knowledge that AI can be used more productively as a development aid. This makes it possible to evaluate not only whether the final system improves, but also whether AI is practically useful in a constrained, iterative, technically demanding development

process. The aim is thus not to let AI carry the work, but to investigate whether it can function as a tool for assisting and streamlining the work.

CHAPTER

# THREE

# THE FIRST ITERATION

The implementation of this program started by just writing something that generated something. This was done to not get stuck in the planning phase. You could consider this a contradiction to the part in software design which states that good planning is imperative. However, trying out some code and figuring out what works, what doesn't and what needs to be implemented in the models is very helpful. Therefore the first iteration was a test to get some systems working and finding out what it neccesary to make it work and what is not imporatant.

## 3.1 The iteration process

After the first iteration lots of issues were observed and were put into the spec for the second iteration which improved upon the model in various ways. Some imporatnt features were restricting the range so that an acutal instrument can play it, be that an instrument or the human voice.

The goal of the first iteration was to get a "proof of concept" working. I wanted to test out how I could generate notes and visualize them as well as listen to audio of the notes generated. For this I used *python* to generate *liypond* files. From these files I was able to generate PDFs of sheet music as well as *wave-files* to listen to how it sounds without having to play/sing it myself.

## 3.2 Software architecture

Figure 3.2.1 shows the structure of the first iteration. The pipeline is small and direct: the configuration is hard-coded in the main script, rhythm and pitch are generated independently for each voice, and the result is exported directly to LilyPond without a richer planning or constraint layer.

## 3.3 Implementation

This first iteration was a very primitive handling of generating music. It had very few constraints on the melody and contained little to no metadata on the music which made it unsuitable for actual music.

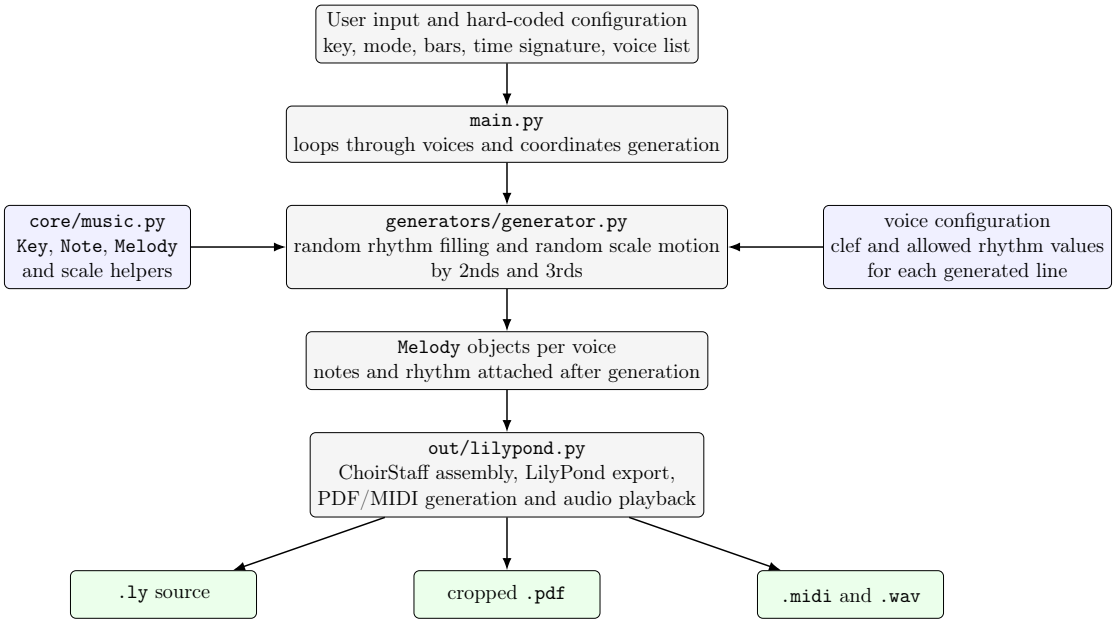



**Figure 3.2.1:** Architecture of the first iteration melody generator. The first proof of concept uses a small script-driven pipeline where rhythm and pitch are generated per voice and then exported directly through LilyPond.

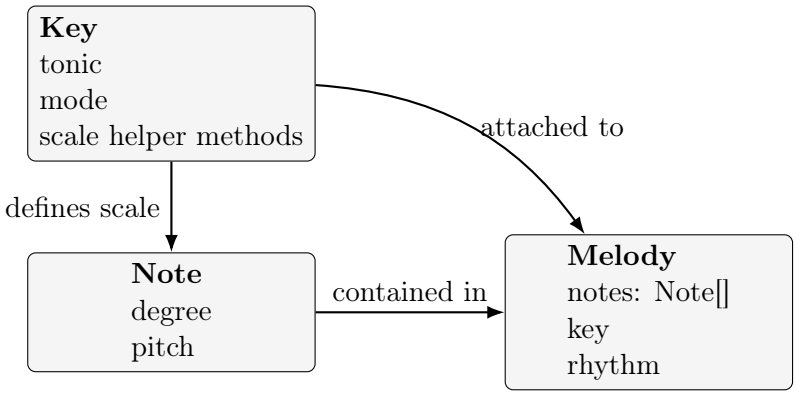



**Figure 3.3.1:** Simplified object model of the first iteration. The representation is enough to store notes in a key and export a melody, but it does not yet describe motifs, bars, phrases, harmony, or form.

Figure 3.3.1 shows that the first iteration really only has the minimum amount of musical structure needed to get notes onto a staff. A note knows its scale degree and pitch, and a melody is mainly a list of notes tied to a key. This is enough for rendering, but it does not yet give the program any good way to reason about motifs, phrase endings, harmonic context, or the usable range of a voice.

A good example of this can be seen in the note generator itself. The code below shows that the melody is made by moving randomly through a small list of intervals, while keeping track only of the current position in the scale.

```
def generate_notes(num_notes=4, key: Key = Key("c",
    "major")) -> Melody:
      intervals = [-2, -1, 0, 1, 2]  # only 2nds and 3rds
      melody_notes = []
      current_idx = 0
      scale = key.get_scale_pitches()

      for _ in range(num_notes):
          val = random.choice(intervals)
          current_idx = (current_idx + val) % len(scale)
          note = Note(degree=current_idx+1,
              pitch=scale[current_idx])
          melody_notes.append(note)

      return Melody(notes = melody_notes, key = key,
          rhythm=None)
```

The code generated the notes within a given scale and restricted the distance (called interval in music theory) to only 2nds and 3rds. However this meant that the melody meandered without any other restrictions. If the generated music was long it would have a tendency to go beyond the range of an instrument or the human voice.

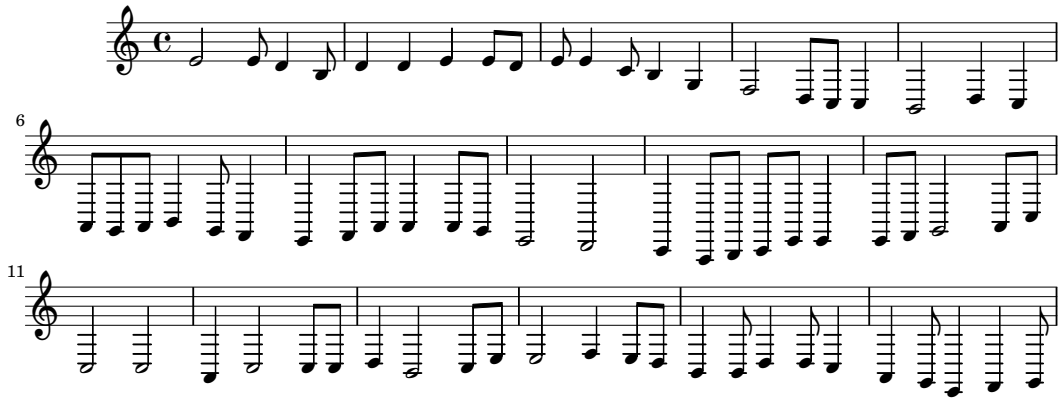
This small function captures both the strength and weakness of the first iteration. It is simple and easy to understand, and it does produce notes that stay inside the chosen key. At the same time, the generator has no memory of larger musical structure. It does not know whether it is at the beginning or end of a phrase, it does not aim for a cadence, and it does not attempt to re-use material in a meaningful way.

## 3.4 Results

The melody generation works fine for smaller melodies, however there is no self-similarity and no rests. This means that the melodies we generate are not particularly musical or interesting, but it is a great starting point to getting to a fully fledged melody generator. In Example 3.4.3 we can see how a short melody using the first iteration looks.

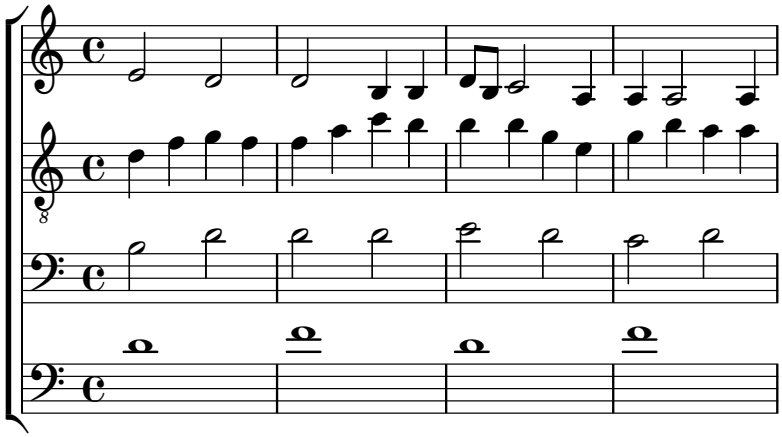
In Example 3.4.1 you can see what happens when the generated melody is longer, it will tend to go beyond what an instrument will be able to play. You can see this clearly in *bar 6 through 16* (the entire 2nd and 3rd system), where there

are very long ledger lines. Some of these notes are not even available on the grand piano.



**Example 3.4.1:** Melody showing how the octave is not restricted and the melody wanders outside a playable range. Listen in the online archive.

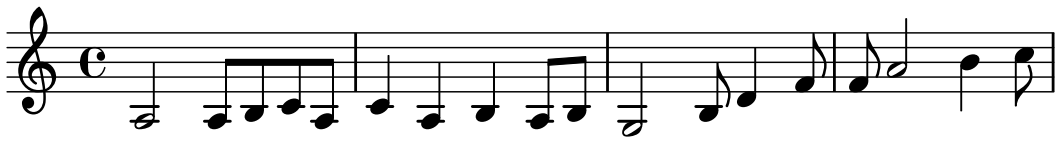
However a good feature that was included was the ability to generate more voices at the same time. This feature was quite primitive since the voices were generated independently and did not adhere to any harmony, however it can be a good skeleton to use later if we get to the point that we want to actually harmonize with the melody we generate. In Example 3.4.2 you can see how we generated four voices all with their own rhythmic restriction. The bottom voice only has whole notes, the second lowest one has only half-notes, the second highest has only quarter and the top voice has a combination of several types.



**Example 3.4.2:** Melody showing how we can generate multiple voices with different rhythmic restrictions. However there is no coherent harmony here every note is placed without regard of the other notes. Listen in the online archive.

## 3.5 Discussion

As we can see the first iteration shows a promising proof of concept, however it is quite limited and needs lots of improvement before we have a good system for



**Example 3.4.3:** Example melody of the first iteration melody generator. This follows the simple rule of no large jumps, but follows no meaningful structure or phrasing. Listen in the online archive.

melody generation.

For the second iteration we need to be able to enforce more restictions on the melody so that it follows more convetions of good melody writing. Additionally we need to have some self-similarity so that the music follows some kind of musical form in order for it to feel coherent and part of the same idea.
The second iteration will focus on creating a generation algorithm that tat is able to generate a melody with a structure. Meaning when generating a melody we will generate a phrase, that phrase should contain an idea, a repetition, and the end of the melody should end resolved with a long note on the tonic of the key. Additionally we need to restirct the range of the melody so that it will be playable/singable.

## 3.6 Goals for next iteration

The main goal for the second iteration is to improve the modelling of the musical concepts like motifs, phrases, musical notes and duration as well as the keys and scales. If we have a better and more robust model for these concepts the implementation of the melodic and harmonic rules will be much easier. Additionally it will be easier to maintain as the current iteration is quite hard to read and understand.

CHAPTER

# FOUR

# THE SECOND ITERATION

As discussed in the previous section there are several issues with the first iteration the second iteration aims to solve. Most of the improvements over the first iteration are not seen by the end-user as the bulk of the improvement is the code handling the musical objects.

## 4.1 Software architecture

Figure 4.1.1 shows the second iteration in the same architectural style as the other iterations. Compared with the first iteration, the generation now passes through explicit musical objects such as `Motif`, `Bar`, and `Phrase`, and the generator reuses motivic material instead of producing every event independently.

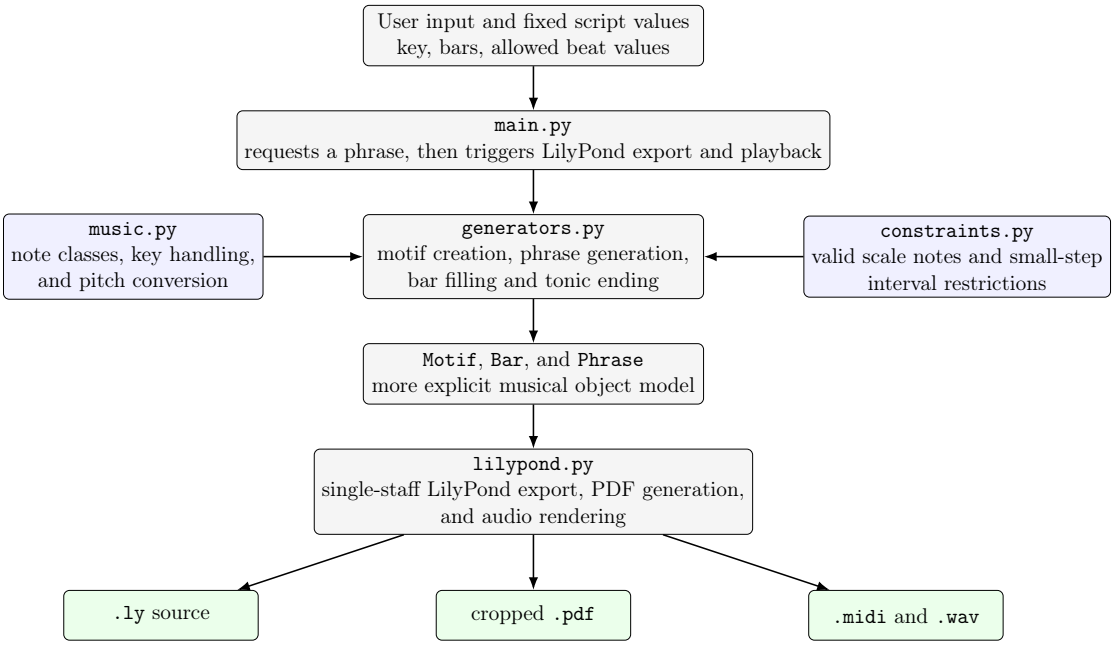



**Figure 4.1.1:** Architecture of the second iteration melody generator. Motivic material and phrase-level objects make the generator more structured than the first iteration, even though the constraint handling is still lightweight and local.

## 4.2 Implementation

As mentioned the code structure has been improved in this iteration, below you can see an excerpt form the code in the *music.py* file.

```
class Note:
    def __init__(self, degree, key, midiNum, duration):
        self.degree = degree
        self.key = key
        self.midiNum = midiNum
        self.duration = duration

class Motif:
    def __init__(self, key, beats=[], notes=[]):
        self.key = key
        self.beats = beats
        self.notes = notes

    def getLength(self):
        length = 0
        for beat in self.beats:
            length += beat
        return length

class Bar:
    def __init__(self, key, beats, notes=[]):
        self.beats = beats
        self.notes = notes

class Phrase:
    def __init__(self, key, bars):
        self.bars = bars
        self.key = key
```

Comparing to the first iteration this has a more defined structure and easier to read, however it needs improvement.

Figure 4.2.1 shows the same development at a slightly higher level. The important step is that the melody is no longer treated only as a flat stream of notes. By introducing objects such as `Motif`, `Bar`, and `Phrase`, the code starts to reflect the way a melody is grouped in actual music.

The second iteration improved upon the structure of the melody. By using a motif we forced some self-similarity in the generated melody which helped generating something more musical, however this generation is very dependent on the generated motif being musical.

Structurally the second iteration is quite similar to the first iteration however, the objects used for handling the melodies and such are more robust and easier to understand. The most important improvement is therefore not a large increase in theoretical correctness, but that the representation now makes it possible to talk about repeated material and phrase structure in the code itself.

At the same time, the objects are still quite lightweight. A `Note` stores a MIDI

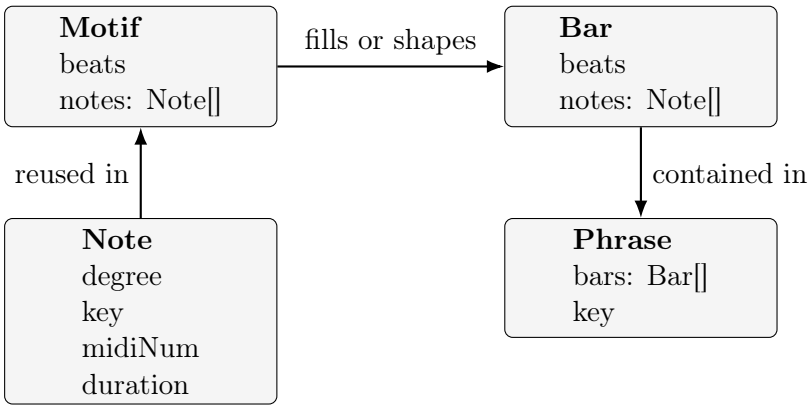
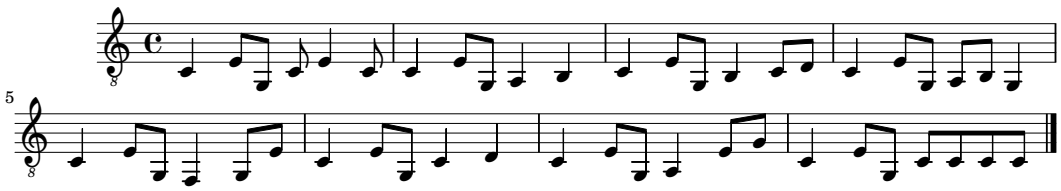



**Figure 4.2.1:** Simplified object model of the second iteration. The important change is the introduction of explicit musical groupings such as motifs, bars, and phrases, which makes repetition and phrase-level planning possible.

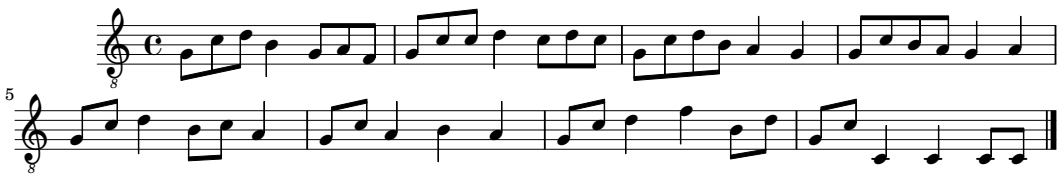
number and a duration, but there is still no explicit harmonic plan, no voice profile, and no event metadata for things such as rests or chromatic adjustments. This means that the representation is better than in the first iteration, but still too limited to support the kinds of transformations and constraints that are needed in the final system.

## 4.3 Results

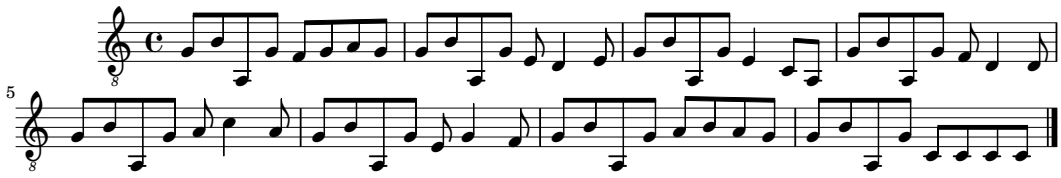
We can see in Example 4.3.1 that we have a more defined structure. The generator works by generating a motif which is then re-used in every measure in the piece. The rest of the measures are filled with random notes moving by 2nds or 3rds to make a somewhat coherent and singable melody.



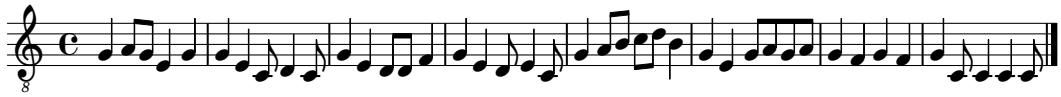
**Example 4.3.1:** Demonstration of a melody generated using the second iteration. Listen in the online archive.



**Example 4.3.2:** A second generated melody from the second iteration, showing the same motif-reuse strategy with a different random outcome. Listen in the online archive.



**Example 4.3.3:** A third generated melody from the second iteration, again reusing an initial motif across the phrase. Listen in the online archive.



**Example 4.3.4:** A fourth generated melody from the second iteration, illustrating how the same method can still lead to noticeably different melodic surfaces. Listen in the online archive.

The generated melodies are still quite random and does not follow any musical form or adhere to melodic restrictions other than that the melody has smaller steps and not big leaps.

## 4.4 Discussion

The most glaring issue with this iteration is that the melodic rules have not been formalized and implemented as code. Generating a melody using this restricted form of generation works well when the generated motif is good, and with the current implementation this only happens randomly. By implementing melodic rules and restrictions this generator will greatly improve the melody generation.

In the first iteration also had the ability to generate several voices at the same time, however, this will not be re-implemented until the melodic generation has been improved and we can enforce harmonic restrictions on one voice. If this is achieved then multi-voice generation will become much more interesting.

Additionally the second iteration does not properly spell all notes in the different keys, this does not affect the generation in any way, though it is a trivial fix it was not implemented since the structure of the second iteration was incompatible with it. Therefore it will only be implemented in the final iteration.

We would also like to implement augmentation of a given melody or phrase, meaning the ability to transpose it up or down or use the same rhythmic outline and moving the notes around to follow a new chord. This will help develop the generated motif and melody to follow a more structured outline of a well composed melody. A key to a good melody is self-similarity, the second iteration just has repetition of a motif, not variations of it.

## 4.5 Goals for the next iteration

The main goals of the third iteration is to be able to augment already generated melodies and be able to enforce constraints on the generation. To ble able to easily augment the melodies and phrases we need a robust model for the melodies made of good objects. Well designed objects is also key to ble able to enforce constraints.

CHAPTER

# FIVE

# THE THIRD ITERATION

For the third iteration the use of AI was implemented.

## 5.1 Use of AI

The third iteration is the first stage of the project where AI is used directly as part of the development workflow. This choice was deliberate. By first developing two iterations without AI assistance, the project established a clearer understanding of the domain, the weaknesses of earlier designs, and the kinds of musical and architectural decisions that had to remain under human control. This made it possible to use AI more critically and productively in the third iteration, rather than relying on it before the problem itself was properly understood.

Codex was chosen as the main AI tool because it is designed specifically for software engineering work rather than only general chat interaction [36, 37]. In practical terms, this means it is suited to tasks such as reading an existing code base, proposing or implementing code changes, helping with refactoring, and supporting documentation and debugging in context. These capabilities are especially relevant in a project like this one, where the challenge is not merely to generate isolated code fragments, but to evolve a structured system over several iterations.

The rationale for using Codex in this thesis is therefore not that it replaces software engineering judgment. Rather, it is used as a development aid inside an already established iterative process. The intention is to use AI as a tool for acceleration, exploration, and implementation support, while ensuring that architectural choices, musical decisions, and validation remain the responsibility of the developer. In that sense, Codex is treated as a collaborator within the workflow, not as a crutch that removes the need for understanding the system being built.

## 5.2 Software architecture

In Figure 5.2.1 we can see how the system is built and how the different modules interact. Compared with the earlier iterations, the third iteration is no longer organized around one large entry script. The command-line workflow has been moved into its own module, while the main entry point is used to define the default

configuration of the generator, such as the weighting of the soft constraints. This makes the architecture easier to understand and easier to adapt for experiments.

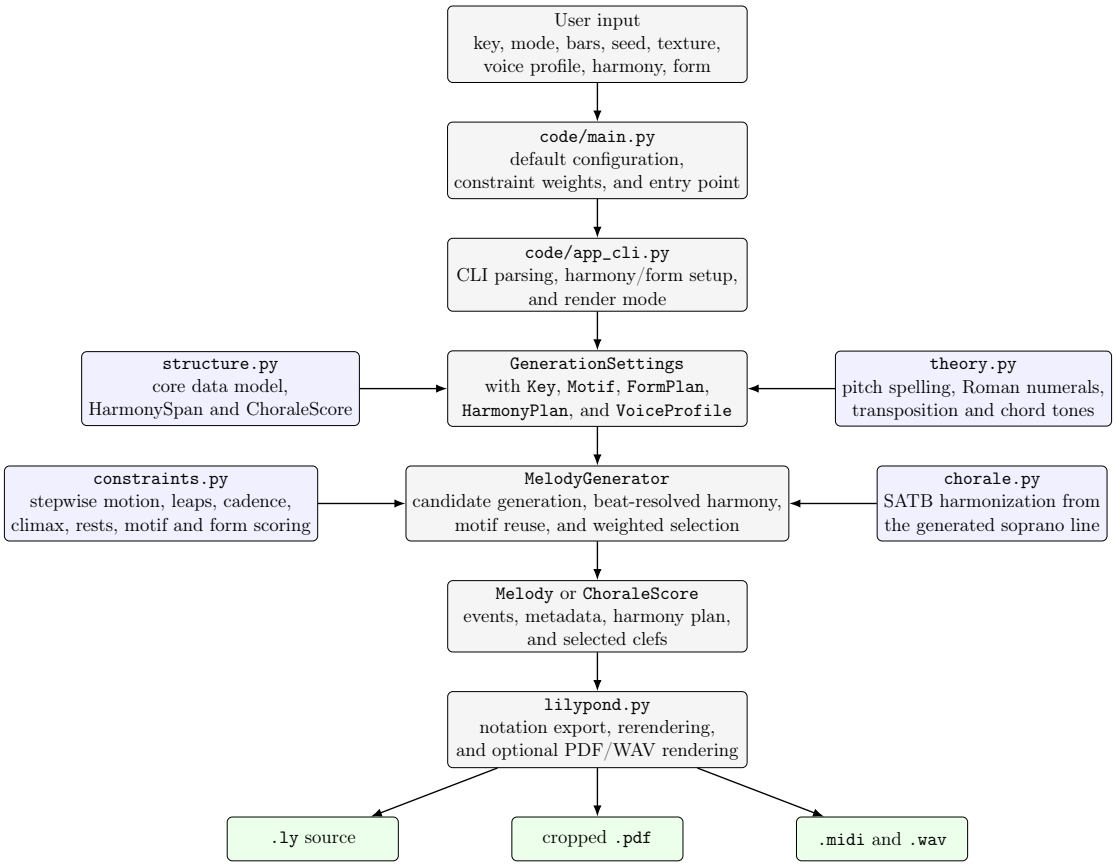



**Figure 5.2.1:** Architecture of the third iteration melody generator. The reusable CLI layer in `app_cli.py` handles parsing and workflow concerns, while `main.py` defines the default constraint configuration. The generator then combines beat-resolved harmony with a soft-constraint stack, and the resulting melody or chorale is exported through LilyPond into notation and audio assets.

## 5.3 Implementation

The most important change in the third iteration is not only that the system got larger, but that the internal representation became detailed enough to support actual musical reasoning. In the first iteration the program mainly handled notes inside a key, and in the second iteration it added motifs, bars, and phrases. In the third iteration, the object model was extended further to include explicit event objects, form sections, harmony spans, and voice profiles. This gives the generator a much clearer description of the musical situation it is working in.

The code excerpt below shows a more representative part of that change. Instead of only storing notes and durations, the third iteration introduces objects that describe vocal range, formal function, and motivic material directly in the data model.

```
@dataclass(frozen=True)
class VoiceProfile:
```

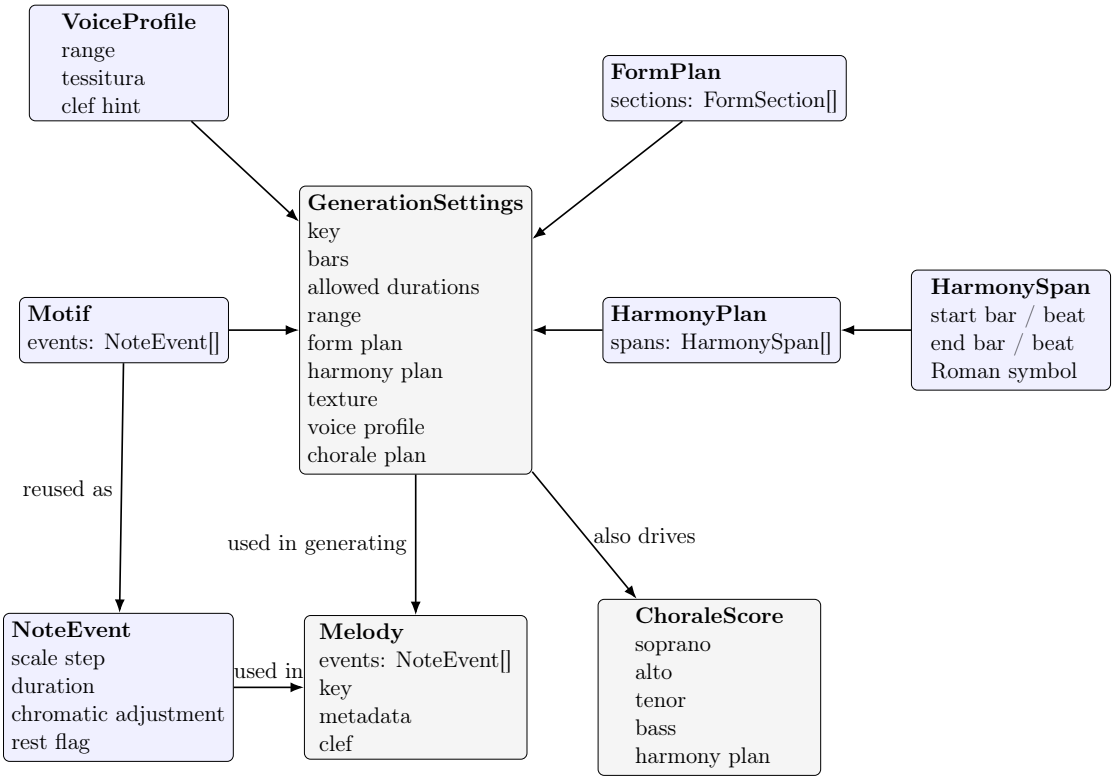



**Figure 5.3.1:** Simplified object model of the third iteration. Compared with the earlier iterations, musical information is no longer limited to notes and phrases, but also includes beat-resolved harmony spans, texture settings, voice profiles, rests, and a separate chorale result type.

```
    name: str
    range_min: int
    range_max: int
    tessitura_min: int
    tessitura_max: int
    clef_hint: str | None = None

    @property
    def melodic_span(self) -> int:
        return self.range_max - self.range_min


@dataclass(frozen=True)
class FormSection:
    label: str
    start_bar: int
    end_bar: int
    role: str
    source_bar: int | None = None
    transform: str = "free"


@dataclass(frozen=True)
class FormPlan:
    kind: str
    sections: tuple[FormSection, ...] = ()

    def section_for_bar(self, bar_number: int) -> FormSection | None:
        for section in self.sections:
            if section.start_bar <= bar_number <= section.end_bar:
                return section
        return None


@dataclass(frozen=True)
class ChoralePlan:
    voice_profiles: tuple[VoiceProfile, ...] = ()


@dataclass(frozen=True)
class Motif:
    events: tuple[NoteEvent, ...]
    name: str = "motif"

    @classmethod
    def from_steps(cls, scale_steps: list[int], durations: list[float], name:
        str = "motif") -> "Motif":
        if len(scale_steps) != len(durations):
            raise ValueError("Motif␣scale␣steps␣and␣durations␣must␣have␣equal␣
                length")
        return cls(
            events=tuple(
                NoteEvent(scale_step=scale_step, duration=duration)
                for scale_step, duration in zip(scale_steps, durations)
            ),
            name=name,
        )
```

```
@property
def length(self) -> float:
    return sum(event.duration for event in self.events)

def transpose_diatonic(self, step_shift: int) -> "Motif":
    return Motif(
        events=tuple(event.transpose_diatonic(step_shift) for event in
            self.events),
        name=self.name,
    )
```

This is a more useful view of the third iteration than a very small note object on its own. `VoiceProfile` gives the generator a way to talk about range and tessitura, `FormSection` and `FormPlan` describe larger formal roles such as repetition and cadence, and `Motif` makes it possible to store and transform a musical cell as an object instead of only reproducing it procedurally. Taken together, these structures show that the third iteration is not just generating more notes, but reasoning about richer musical objects.

Another important improvement is that the generation task is gathered into a single settings object. Rather than relying on hard-coded assumptions spread across one large script, the third iteration collects key, form, harmony, durations, range, and voice information before generation starts, while leaving the default weighting of the musical constraints in the main entry point. This makes the system easier to extend and also makes the constraints easier to apply consistently.

It is also useful to follow what the system actually does after a command is received. The first step is shown in Figure 5.3.2. The command line is parsed in `app_cli.py`, and the separate user choices are then collected into a single `GenerationSettings` object. This is where the key, time signature, form plan, harmony plan, motif, texture, and voice profile are brought together into one consistent description of the task.

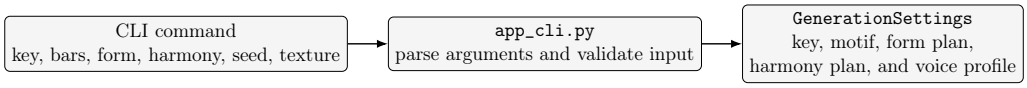



**Figure 5.3.2:** The first stage of the third iteration. The command line is parsed and translated into a single structured settings object before any music is generated.

Once the settings are ready, the melody generator begins to prepare the musical context before it commits to any pitches. Figure 5.3.3 shows this middle stage. The generator first creates a rhythmic skeleton that fits the number of bars and the phrase layout. It then decides where the motif should return and where the melodic climax should be placed. Only after those larger targets are set does it begin to choose notes.

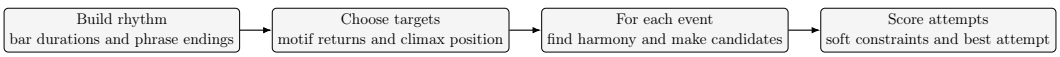



**Figure 5.3.3:** The melody-generation stage in the third iteration. The generator first decides structural targets, then evaluates candidate notes under the active harmonic and formal context.

Figure 5.3.4 illustrates this idea using the opening of one real third-iteration output. At this stage the interesting information is not yet the pitch contour, but the rhythmic shape and the bar structure. The generator has already committed to note lengths and phrase pacing, which means the later pitch decisions will happen inside a more stable framework.

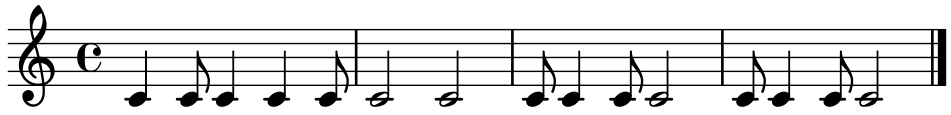


**Figure 5.3.4:** A rhythmic skeleton corresponding to the first four bars of a generated third-iteration melody. At this stage the durations and bar structure are fixed, while the actual pitch content is still to be chosen.

The pitch generation itself happens as a repeated local decision process. For each event position, the code looks up the current bar, beat, harmony span, and form role, and then builds a group of candidate notes around the existing melodic context. These candidates are scored by the soft-constraint stack, which considers things such as stepwise motion, leap recovery, harmonic fit, motif similarity, cadence behaviour, and the general melodic shape. Rather than simply walking around the scale, the third iteration repeatedly asks what would make musical sense at this exact point in the phrase.

The next code excerpt shows that process more directly. It is short, but it captures the main difference from the previous iterations: the note choice is made inside an explicit musical context containing beat position, harmony, motif targets, and form role.

```
for index, duration in enumerate(rhythm):
    bar_number, beat_in_bar = beat_positions[index]
    harmony_span = self.settings.harmonic_plan.chord_for_position(
        bar_number,
        beat_in_bar,
        self.settings.time_signature.bar_length,
    )
    motif_target_step = motif_targets.get(index)
    section = self.settings.form_plan.section_for_bar(bar_number)
    context = CandidateContext(
        key=self.settings.key,
        events=tuple(events),
        index=index,
        bar_number=bar_number,
        beat_in_bar=beat_in_bar,
        total_events=len(rhythm),
        climax_index=climax_index,
        climax_step=climax_step,
        phrase_end_bars=phrase_end_bars,
        current_duration=duration,
        harmony_span=harmony_span,
        motif_target_step=motif_target_step,
        section_role=section.role if section is not None else "free",
        section_transform=section.transform if section is not None else
            "free",
    )
    candidate_steps = self._candidate_steps(
```

```
            events,
            index,
            climax_index,
            climax_step,
            motif_target_step,
            context,
        )
        chosen_note, chosen_score = self._choose_candidate(candidate_steps,
            context)
```

The result of that process is shown in Figure 5.3.5. Here the same rhythmic outline has been given actual pitches, and the opening four bars now begin to show the musical character of the line. This makes the implementation easier to follow because it shows how the final melody is not generated all at once, but emerges from rhythm, context, and repeated local choices.

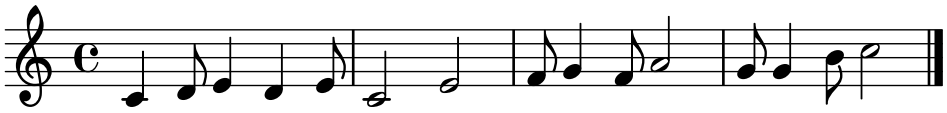


**Figure 5.3.5:** The same opening after pitch generation. The rhythmic framework has now been turned into a concrete melody by repeated candidate evaluation under the active harmonic and formal context.

The final melody is selected from several complete attempts rather than from one single pass. After that, the workflow splits as shown in Figure 5.3.6. If the requested texture is just a melody, the resulting line is exported directly. If the texture is chorale, the melody is treated as a soprano line and passed to the harmonizer, which chooses alto, tenor, and bass notes according to the same harmony plan and the SATB voice profiles. The last step in both cases is export through LilyPond, with optional PDF, MIDI, and WAV rendering.

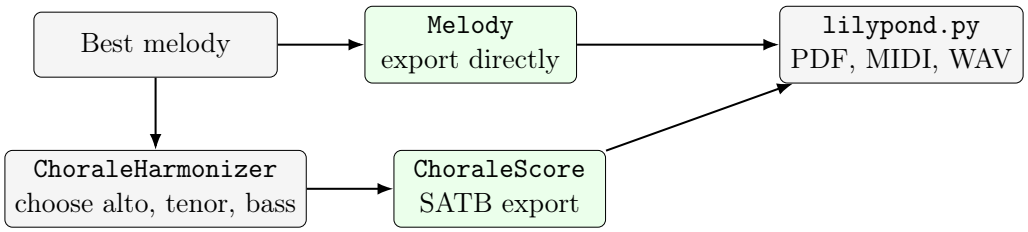



**Figure 5.3.6:** The final stage of the workflow. The best melody can either be exported directly or used as soprano input for chorale harmonization before rendering.

Figure 5.3.7 shows the final branch where the melody is no longer kept as a single line, but is instead used as the soprano basis for a simple SATB-style texture. This figure is especially useful because it shows that the third iteration does not only improve the quality of isolated melodies, but also provides enough harmonic and structural information to support a later harmonization stage.

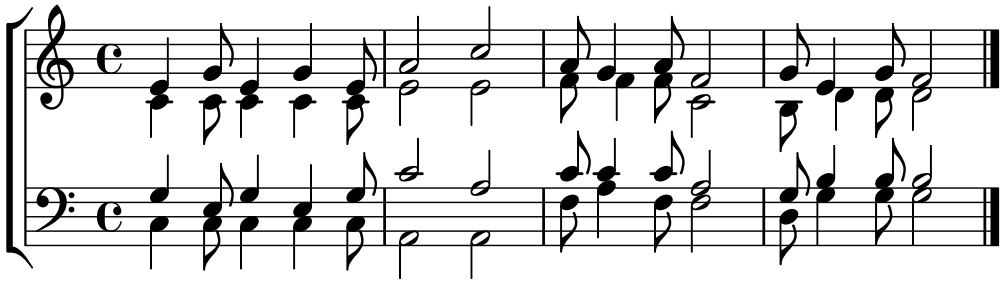


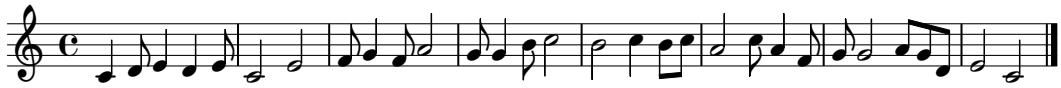
**Figure 5.3.7:** The same musical material after chorale harmonization. The generated melody is treated as a soprano line, and alto, tenor, and bass voices are chosen against the same harmonic plan.

## 5.4 Results

The musical output of the third iteration is noticeably more controlled than in the previous versions. The melodies remain within a defined vocal or instrumental range, the generator can take a harmonic plan into account, and motif reuse is no longer limited to literal repetition. The addition of form planning also makes it possible to think in larger units such as sentences and periods rather than only local note-to-note movement.

Rather than presenting many examples that differ only in seed, it is more useful to structure the results by capability. The examples below therefore act as a compact version of the appendix: one example for a basic melody result, one for explicit form handling, one for rhythmic restriction, one for alternative choral scoring, and one final longer example showing what can be done when the user gives the system a more detailed harmonic plan.

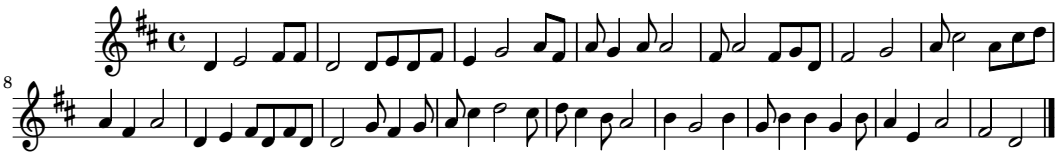
The first capability is simply that the generator now produces a usable single-line melody. Example 5.4.1 shows an ordinary output in C major using the default settings. Compared with the first two iterations, the important point is not that the melody is perfect, but that it remains within range, reuses material in a more deliberate way, and arrives at a cadence that feels shaped rather than accidental.



**Example 5.4.1:** A melody generated by the third iteration in C major. This example uses seed 2000 and shows a sentence-like shape with clear motivic reuse. Listen in the online archive.

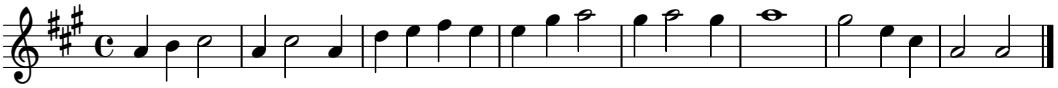
The second capability is explicit form control. This becomes easier to observe when the melody is allowed to unfold over a larger span. Example 5.4.2 shows a 16-bar melody in D major generated using the `period` form plan. In this case the balanced phrase structure and the different treatment of the return toward cadence are easier to recognize than in the shorter examples, and this makes it clearer that the generator is no longer only producing note-to-note movement, but is instead working with a larger formal outline.

The third capability is rhythmic restriction. The generator can be given a narrower rhythmic vocabulary, which changes the surface character without changing



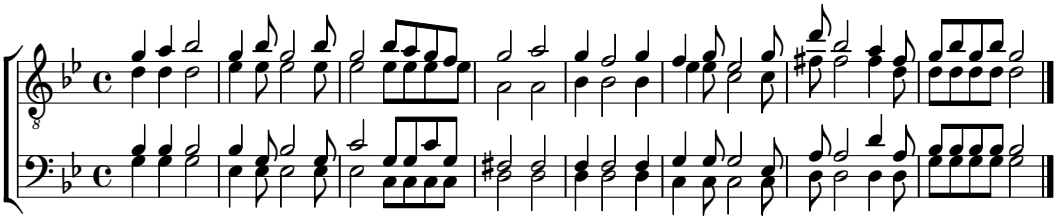
**Example 5.4.2:** A 16-bar melody generated by the third iteration in D major using the explicit `period` form plan and seed 4510. Listen in the online archive.

the larger harmonic and formal workflow. Example 5.4.3 shows a melody in A major generated using only whole, half, and quarter notes. This is useful because it demonstrates that the rhythmic content can be shaped deliberately instead of always relying on the default mixture of durations.



**Example 5.4.3:** A melody in A major generated using only whole, half, and quarter notes with seed 4511. Listen in the online archive.

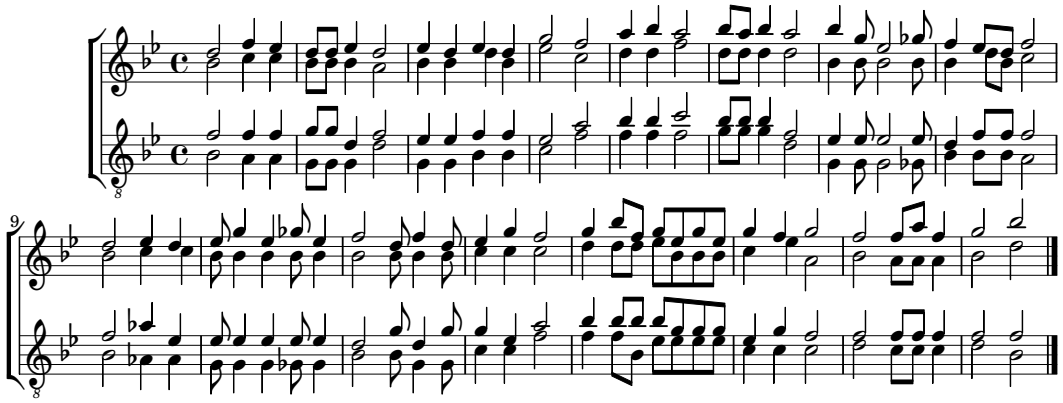
The fourth capability is flexible scoring. The same harmonization workflow does not have to be limited to a standard SATB arrangement. Example 5.4.4 shows a four-part result in G minor using a TTBB layout. This demonstrates that the harmonization layer can be adapted to other ensemble requirements, as long as appropriate voice profiles are provided.



**Example 5.4.4:** A four-part TTBB-style example generated by the third iteration in G minor using seed 4512. Listen in the online archive.

The final capability is explicit harmonic control. This is perhaps the clearest example of how the program can be used differently by someone with more musical knowledge. Instead of relying on the automatic tonal plan, the user can specify a progression directly and use the compact harmony syntax to shape the result bar by bar and beat by beat. Example 5.4.5 shows a longer 16-bar chorale in B♭ major using an explicit progression with borrowed chords such as `iv` and `bVII`. This kind of input allows the output to move beyond the most standard diatonic patterns and makes the system feel more like a compositional tool than only a generator of default examples.

Another clear improvement is that the notation workflow became much more usable. The system can now place generated files in seed-based output folders, choose a clef more intelligently, and optionally render cropped PDF files and audio



**Example 5.4.5:** A 16-bar harmonized example generated by the third iteration in B♭ major using an explicit harmonic progression with the borrowed `iv` and `bVII` chords. This is the kind of output that becomes possible when the user approaches the program with a more deliberate harmonic idea. Listen in the online archive.

from the same command-line call. This does not change the melody itself, but it makes the generator much easier to use as part of a real compositional and thesis-writing workflow.

Even with these improvements, the results are still heuristic rather than guaranteed. The generator is better at preferring musically convincing solutions, but it does not prove that every melody will be good. The value of the third iteration is therefore that it makes good results much more likely while also making the failures easier to understand and improve. The appendix can be read as an extended version of this results section, where the same capabilities are shown across more keys, more rhythm settings, and more texture choices.

## 5.5 Discussion

The third iteration is the first version where the musical objects and the generation strategy point in the same direction. In the earlier iterations the generator could produce output, but the internal representation was too weak to carry much musical meaning. Here, the object model is rich enough that concepts such as form, harmony, range, and motif transformation can be represented directly in the code instead of only being implied by the generation procedure.

This is also the first iteration that feels like a reusable foundation rather than only an experiment. Because the settings, structure, theory, and constraint layers are separated more clearly, the system can be extended without rewriting the entire program. That is especially important for future work such as more robust four-part writing, since multi-voice generation requires a representation that can handle range, harmonic function, and voice-leading information at the same time.

The third iteration is still not a complete compositional system. The constraint weights are chosen heuristically, and musical quality still depends on how well those constraints are balanced. However, compared with the first two iterations, the final version is much closer to a tool that can be discussed, refined, and extended in a systematic way rather than only adjusted by trial and error.

CHAPTER

# SIX

# DISCUSSION

This section will discuss the main observations of the project. I will discuss the process of iterative development and how it influenced the final system, the use of AI in the development process and whether or not it is a useful tool for programmers. The final system, its structure and the generated music will also be discussed. Comparison of the three iterations will also be made to show the progression from a simple script to a modular and musically aware system. Finally I will look at some of the limitations of the current system and possible directions for future work.

## 6.1 Iterative development

The iterative development process has been very useful in this project. Rather than trying to solve the entire problem at once, each iteration solved a narrower version of it and made the weaknesses of that solution easier to see. This was especially important in a project like this one, where the quality of the system is not only a question of whether the code runs, but also whether the generated music is usable and convincing.

Figure 6.1.1 gives one view of how the three iterations developed over time. The first iteration reached a moderate size quickly because it was focused on a small proof of concept. Once it could generate notes, render LilyPond output, and produce audio, it had already answered its main question. The second iteration grew more slowly and remained smaller overall, because its main purpose was not to add many new subsystems, but to improve the internal handling of musical material. The third iteration grew much more rapidly because it brought together a richer data model, explicit constraint handling, harmony support, form planning, rendering workflows, and later SATB-compatible extensions.

This growth pattern is important because it shows that the final system did not appear fully formed. The first iteration established that automatic generation and rendering were feasible. The second iteration showed that more musical output required better abstractions and some form of self-similarity. The third iteration was where those lessons were formalized into a larger system. In that sense, the value of the early iterations is not only the code they produced, but also the questions they made visible.

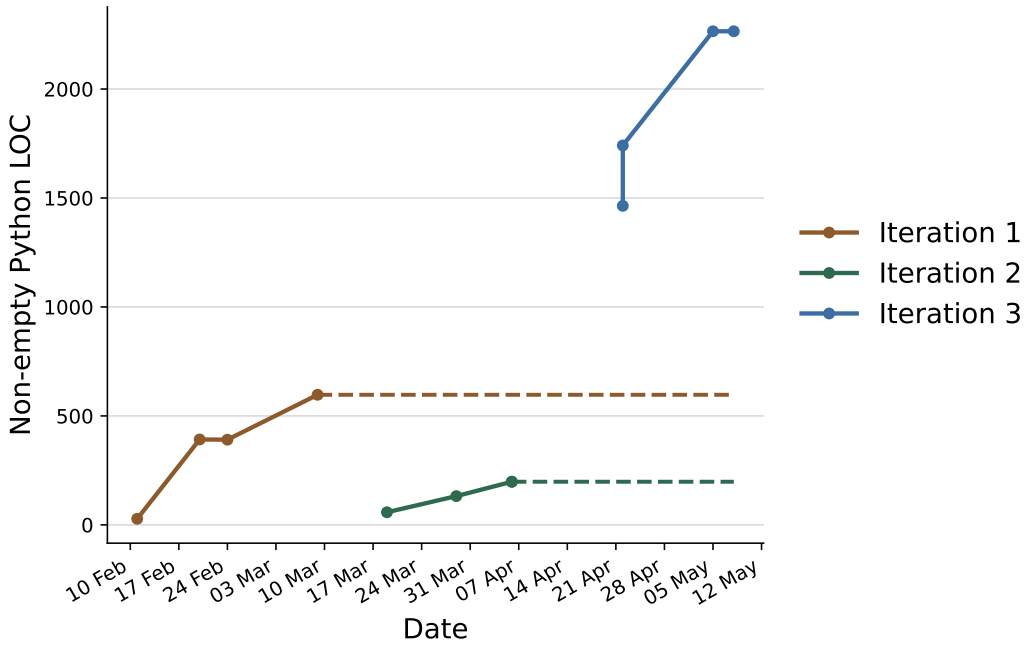



**Figure 6.1.1:** Growth of the three iterations over time measured as non-empty Python source lines in the iteration-specific code. Dashed segments indicate that an iteration had been frozen while the thesis work continued.

Table 6.1.1 summarizes the practical differences between the iterations. The main observation is that the first two iterations each solved only part of the problem. The first iteration had rendering and even rudimentary multi-voice support, but the melodies themselves were musically weak. The second iteration improved motivic handling and made the code easier to reason about, but it still lacked explicit theory-aware constraints. Only the third iteration combines musical structure, reusable abstractions, harmony handling, and a workflow that is robust enough to support the thesis examples in a reliable way.

## 6.2 AI in development

The use of AI had a clear influence on the development process, but that influence was strongest when it was used after a good understanding of the problem had already been established. By the time the third iteration started, the earlier iterations had already shown which parts of the generator were weak, which musical ideas were difficult to encode, and which parts of the software structure needed to change. This made it possible to use AI more critically and with better prompts than would have been possible at the start of the project.

In practice, AI was most useful when it was treated as a development aid rather than as a source of complete solutions. It was helpful for refactoring, for exploring alternative implementations, for surfacing edge cases, and for speeding up repetitive tasks that would otherwise take a disproportionate amount of time. This aligns well with the broader picture described in the software-development literature, where AI tools are increasingly common, but still require human oversight and validation [31, 32, 33, 34, 38].

| Feature | Iteration 1 | Iteration 2 | Iteration 3 |
|---|---|---|---|
| LilyPond export and audio rendering | yes | yes | yes |
| Independent multi-voice output | yes | no | yes |
| Explicit motif reuse | no | yes | yes |
| Phrase or form planning | no | limited | yes |
| Automatic tonal harmony plan | no | no | yes |
| Roman-numeral harmony input | no | no | yes |
| Melodic soft constraints | no | limited | yes |
| Rest handling in melody generation | no | no | yes |
| Register-aware voice profiles and clefs | no | no | yes |
| SATB-compatible harmonization path | no | no | yes |
| Descriptive CLI and reproducible seed workflow | limited | limited | yes |

**Table 6.1.1:** Comparison of the main musical and technical capabilities across the three iterations.

At the same time, the project also showed some of the limitations of using AI in this kind of work. A generator for music is not only a software problem, but also a domain problem. It is therefore not enough that the code compiles or that the architecture looks reasonable. The generated melodies must also make musical sense. This means that the usefulness of AI depends strongly on the developer being able to evaluate both the software and the musical result. If that understanding is missing, it becomes difficult to distinguish between code that is merely plausible and code that actually improves the system.

For this reason, the use of Codex in this project was most valuable when it accelerated work that was already framed by the developer, not when it attempted to replace the framing itself. The thesis therefore supports a fairly balanced view of AI in development: it can be an effective tool for implementation and iteration, but it does not remove the need for understanding, evaluation, and responsibility on the part of the developer.

## 6.3 The system structure

The progression between the iterations is not only visible in the generated output, but also in the internal structure of the software. Figures 3.2.1, 4.1.1, and 5.2.1 show a clear movement from a small script-driven pipeline toward a more modular and theory-aware design.

Table 6.3.1 shows that the main architectural improvement is not simply that the third iteration has more files or more code. The important change is that it has clearer boundaries between responsibilities and more explicit musical objects.

| Aspect | Iteration 1 | Iteration 2 | Iteration 3 |
|---|---|---|---|
| Main representation | note and melody objects with limited metadata | motif, bar, and phrase objects | settings, motif, form, harmony, melody, and chorale-related objects |
| Generation strategy | random local note motion with separate rhythm generation | motif-first phrase generation with small-step filling | weighted search with explicit candidate scoring and harmonic context |
| Harmony handling | none | none | manual Roman numerals or automatically generated tonal plan |
| Module structure | small script plus helper modules | clearer separation, but still tightly coupled | dedicated theory, structure, constraints, generator, rendering, and CLI layers |
| Extensibility | difficult to extend without rewriting | better than iteration 1, but still narrow | designed to support variants, rendering modes, and future multi-voice work |

**Table 6.3.1:** Architectural comparison of the three iterations.

In the first iteration, the generator was close to a single procedural script. In the second iteration, the introduction of motifs, bars, and phrases made the code more understandable, but the structure was still too narrow to support many kinds of extension. In the third iteration, melody generation, theory handling, constraint scoring, rendering, and user input are separated more clearly.

This separation matters because it changes what kinds of improvements are possible. When the generator is too tightly coupled, even small musical changes can require rewriting large parts of the code. When the representation is more explicit, it becomes possible to work on one problem at a time. For example, clef selection, harmony parsing, or form planning can be improved without changing the entire generation pipeline. That makes the third iteration much more suitable as a foundation for future work.

Another important structural change is the development of the musical objects themselves. In the first iteration, the code mainly needed to store notes in a key and send them to LilyPond. In the second iteration, the representation became more phrase-oriented through motifs, bars, and phrases. In the third iteration, the object model became rich enough to describe not only notes, but also rests, chromatic adjustments, harmony spans, voice range, and formal sections. This is one of the main reasons the third iteration can support more musical reasoning than the earlier versions.

## 6.4 The generated music

The quality of the generated music improved significantly between the iterations, but the improvement did not happen all at once. The first iteration showed that the system could generate notes and render them correctly, but the melodies were mostly streams of scale motion with very little larger structure. They often drifted out of a useful register and had little to hold them together beyond staying inside the same key.

The second iteration improved this by introducing a motif-based approach. This gave the melodies more identity, because some material was repeated instead of every note being generated independently. Even so, the result still depended heavily on whether the original motif happened to be good. Since the melodic rules were still only lightly encoded, the generator could create something that looked more structured on the page without necessarily becoming much stronger musically.

The third iteration is where the output becomes more consistently usable. This is mainly because the generator no longer relies only on local interval restrictions or literal repetition. Instead, it can take several kinds of information into account at once: melodic range, harmonic context, phrase role, cadence shape, motif reuse, and soft constraints on motion. This does not make the system perfect, and it does not remove the subjective side of musical evaluation, but it does make the results more controlled and more explainable.

One of the more important observations from the examples is that musical quality in this project is closely tied to representation. When the internal objects are too weak, the output also becomes weak, because the generator simply has too little meaningful information to work with. As the musical objects become richer, the system gains more ways to shape the output in musically relevant directions. This makes the progression between the iterations not only a technical story, but also a musical one.

At the same time, the system still has clear limits. It can generate passable melodies and it can harmonize more than before, but it does not evaluate music in the way a human musician would. The weighting of constraints is still heuristic, and some examples will inevitably work better than others. In that sense, the system should be understood as a structured procedural generator rather than as a complete model of musical composition.

## 6.5 Future work

There are several natural directions for extending the system. The most obvious is more complete four-part writing. The current third iteration already has some SATB-compatible foundations, such as voice profiles, clef handling, harmony-aware generation, and a first chorale path. However, a more mature four-part system would require stronger handling of voice leading, non-chord tones, suspensions, spacing, doubling, and cadence treatment across all voices.

Another useful direction would be stronger evaluation and comparison of outputs. The thesis has mainly assessed results through musical inspection and by checking whether the generated examples follow a set of recognizable melodic and harmonic principles. This is appropriate for the scope of the project, but fu-

ture work could introduce more systematic evaluation, for example by comparing generated outputs against rule-based checklists or user assessments by musicians.

A third direction is more interactive use. The current workflow is already much better than in the first two iterations, especially through reproducible seeds, clearer command-line arguments, and automatic rendering. Even so, the generator is still mainly operated as a batch tool. It would be interesting to develop a more interactive interface where a user can guide harmony, motif transformation, or phrase structure more directly and then immediately listen to the result. This would move the system closer to a compositional aid rather than only a generator.

A major musical point that has been largely ignored in even the final iteration is rhythm. The current system takes into consideration the harmonic context and melodic constraints, however the rhythm is still generated randomly without regard to the musical context. This could be greatly improved by first ensuring that there is a coherence between strong beats and longer notes. This means that faster notes would be more likely on the weak beats, beat 2 and 4. Additionally the rhythm could also be made aware of the harmony, meaning dissonances should be shorter and consonants longer etc. This would make the pulse of the generated music clearar and avoid having short beats on the first beat leading to a feeling of the musical pulse shifting.

We would also take a look at freeing up the generator a bit, in its current state it is quite rigid in the way it generates music which lead to quite similar results between generations. This could be improved by allowing more freedom in for instance the harmonic rhythm and progression. Leading to more varied melodies since they now follow the generated harmonic progression very closely and the harmonic progressions are quite similar since they are adhering to the rules without any freedom.

Finally, the use of AI itself could be explored further. In this thesis, AI was mainly used as a development tool rather than as part of the musical generator. Future work could investigate whether AI could also contribute inside the musical workflow, for example in evaluating outputs, suggesting constraints, or supporting mixed-initiative composition. At the same time, the lessons from this project suggest that such use should still be approached carefully, since the main value lies in supporting informed musical and technical decisions rather than replacing them.

CHAPTER

# SEVEN

# CONCLUSION

This project has shown the development of a procedural music generator. The process utilized iterative development as well as AI assistance in developing the system. This resulted in a modular system which encapsultes musical ideas in an effictive manner which is easy to understand and modify.

The use of AI proved to be a very useful tool to speed up development both in terms of troubleshooting or debugging as well as helping find solutions to the problems you are working on. The issue with AI is that is removes the creative element of programming and simply regurgitates from the ocean of information it has access to. This means that it solves the problems in similar ways they have been done before, this hinders true innovation and creating something completely new.

On the other end we have the problem that AI is very good at generating content, meaning that by prompting AI to solve a problem for us it generates a lot of code. This means that in addition to formulating the problem in such a way that AI can solve it, we also need to go over the code and check that it actually does what we intended it to do, as well as ensuring that it is maintainable and easy to read. In order to do this part effectively we need to understand not only the language we are developing in, but we also need knowledge of the existing code base. For this problem it means that we need to know not only how the language works, but also the language of music theory. If we do not understand this then we would not be able to make any meaningful changes to the code by ourselves, let alone try to prompt engineer without knowing anything about music.

There seems to be a sort of silver lining in the use of AI when developing software. Using it to develop entire systems will lead to the developer not understanding how the code works or having to spend a substantial amount of time reading through the AI-generated code to figure out how it works. Using AI in smaller contexts can yield better results. It is very good at saving time by completing trivial tasks that simply take time, or by debugging small functions that you could have spent minutes or hours tinkering with to solve yourself. This keeps the code your own, but it makes you faster by helping you with the small tasks.

Overall the system generates passable music, there is no defined way to actually grade music objectively, but at least we know that the output follows a few principles associated with good melodic and harmonic structure.

Key takeaways from the project:

- Procedural generation can be used to create music, but it requires a good understanding of the domain and careful design of the system.
- The use of AI in software development can be a powerful tool, but it should be used with caution and in a way that complements human creativity and understanding rather than replacing it.
- The generated music is still heuristic and cannot be evaluated objectively in a complete musical sense, but the final iteration produces results that are substantially more structured, singable, and musically coherent than the earlier versions.

# REFERENCES


[1] *Musikalisches Würfelspiel*. In: *Wikipedia*. Sept. 27, 2025. URL: https://en.wikipedia.org/w/index.php?title=Musikalisches_W%C3%BCrfelspiel&oldid=1313693497 (visited on 03/24/2026).

[2] *Musikalisches Würfespiel (K.516f)*. The Mozart Portal. URL: https://www.mozartportal.com/en/composition/kv-516f-musikalisches-w-rfespiel (visited on 05/13/2026).

[3] Jonas Freiknecht and Wolfgang Effelsberg. "A Survey on the Procedural Generation of Virtual Worlds". In: *Multimodal Technologies and Interaction* 1.4 (Dec. 2017), p. 27. ISSN: 2414-4088. DOI: 10.3390/mti1040027. URL: https://www.mdpi.com/2414-4088/1/4/27 (visited on 05/10/2026).

[4] *Procedural City Generator | 3D City Maker | ArcGIS CityEngine*. URL: https://www.esri.com/en-us/arcgis/products/arcgis-cityengine/overview (visited on 05/10/2026).

[5] Ghassan Hamarneh and Preet Jassi. "*VascuSynth*: Simulating Vascular Trees for Generating Volumetric Image Data with Ground-Truth Segmentation and Tree Analysis". In: *Computerized Medical Imaging and Graphics* 34.8 (Dec. 1, 2010), pp. 605–616. ISSN: 0895-6111. DOI: 10.1016/j.compmedimag.2010.06.002. URL: https://www.sciencedirect.com/science/article/pii/S0895611110000534 (visited on 05/10/2026).

[6] *[1809.04281] Music Transformer*. URL: https://arxiv.org/abs/1809.04281 (visited on 02/26/2026).

[7] Curtis Hawthorne et al. *Enabling Factorized Piano Music Modeling and Generation with the MAESTRO Dataset*. Jan. 17, 2019. DOI: 10.48550/arXiv.1810.12247. arXiv: 1810.12247 [cs]. URL: http://arxiv.org/abs/1810.12247 (visited on 02/26/2026). Pre-published.

[8] Sageev Oore et al. "This Time with Feeling: Learning Expressive Musical Performance". In: *Neural Computing and Applications* 32.4 (Feb. 1, 2020), pp. 955–967. ISSN: 1433-3058. DOI: 10.1007/s00521-018-3758-9. URL: https://doi.org/10.1007/s00521-018-3758-9 (visited on 02/26/2026).

[9] Prafulla Dhariwal et al. *Jukebox: A Generative Model for Music*. Apr. 30, 2020. DOI: 10.48550/arXiv.2005.00341. arXiv: 2005.00341 [eess]. URL: http://arxiv.org/abs/2005.00341 (visited on 02/26/2026). Pre-published.

[10] Ye Bai et al. *Seed-Music: A Unified Framework for High Quality and Controlled Music Generation*. Sept. 19, 2024. DOI: 10.48550/arXiv.2409.09214. arXiv: 2409.09214 [cs]. URL: http://arxiv.org/abs/2409.09214 (visited on 03/24/2026). Pre-published.

[11] Lucas N. Ferreira and Jim Whitehead. *Learning to Generate Music With Sentiment*. Mar. 9, 2021. DOI: 10.48550/arXiv.2103.06125. arXiv: 2103.06125 [cs]. URL: http://arxiv.org/abs/2103.06125 (visited on 03/24/2026). Pre-published.

[12] *Press - No Man's Sky*. URL: https://www.nomanssky.com/press/ (visited on 05/10/2026).

[13] Activision Blizzard. *A Brief History of the World's Smallest First-Person Shooter*. Dec. 20, 2023. URL: https://newsroom.activisionblizzard.com/p/a-brief-history-of-the-worlds-smallest (visited on 05/13/2026).

[14] *CityEngine C++ SDK*. URL: https://esri.github.io/cityengine/cityenginesdk (visited on 05/10/2026).

[15] Gerhard Nierhaus. *Algorithmic Composition*. Vienna: Springer, 2009. ISBN: 978-3-211-75539-6 978-3-211-75540-2. DOI: 10.1007/978-3-211-75540-2. URL: http://link.springer.com/10.1007/978-3-211-75540-2 (visited on 06/03/2026).

[16] Torsten Anders and Eduardo R. Miranda. "Constraint Programming Systems for Modeling Music Theories and Composition". In: *ACM Comput. Surv.* 43.4 (Oct. 18, 2011), 30:1–30:38. ISSN: 0360-0300. DOI: 10.1145/1978802.1978809. URL: https://dl.acm.org/doi/10.1145/1978802.1978809 (visited on 02/26/2026).

[17] *Arc-Consistency Algorithm - an Overview | ScienceDirect Topics*. URL: https://www.sciencedirect.com/topics/computer-science/arc-consistency-algorithm (visited on 05/08/2026).

[18] Maxim Gumin. *Wave Function Collapse Algorithm*. Version 1.0. Sept. 2016. URL: https://github.com/mxgmn/WaveFunctionCollapse (visited on 03/24/2026).

[19] Hao Hua. "A Bi-Directional Procedural Model for Architectural Design". In: *Computer Graphics Forum* 36 (Dec. 1, 2016). DOI: 10.1111/cgf.13074.

[20] Pal Varga and Rafael Bidarra. "Procedural Mixed-Initiative Music Composition with Hierarchical Wave Function Collapse". In: Mar. 26, 2023.

[21] Roland van der Linden, Ricardo Lopes, and Rafael Bidarra. "Procedural Generation of Dungeons". In: *IEEE Transactions on Computational Intelligence and AI in Games* 6.1 (Mar. 2014), pp. 78–89. ISSN: 1943-0698. DOI: 10.1109/TCIAIG.2013.2290371. URL: https://ieeexplore.ieee.org/abstract/document/6661386 (visited on 02/26/2026).

[22] Ken Perlin. "An Image Synthesizer". In: *SIGGRAPH Comput. Graph.* 19.3 (July 1, 1985), pp. 287–296. ISSN: 0097-8930. DOI: 10.1145/325165.325247. URL: https://dl.acm.org/doi/10.1145/325165.325247 (visited on 05/16/2026).

[23] *Random Walk | Stochastic Process, Probability & Diffusion | Britannica*. Apr. 25, 2026. URL: https://www.britannica.com/science/random-walk (visited on 05/16/2026).

[24] Ben. *Cadences - Music Theory Academy - Perfect, Plagal, Imperfect, Interrupted*. Music Theory Academy. Mar. 8, 2013. URL: https://www.musictheoryacademy.com/how-to-read-sheet-music/cadences/ (visited on 05/12/2026).

[25] Arnold Schoenberg. *Fudamentals of Musical Composition*. 1st ed. faber and faber, 1967. 224 pp. ISBN: 0-571-19658-6.

[26] *Sentence (Music)*. In: *Wikipedia*. June 21, 2023. URL: https://en.wikipedia.org/w/index.php?title=Sentence_(music)&oldid=1161220566 (visited on 03/26/2026).

[27] *Period (Music)*. In: *Wikipedia*. Jan. 29, 2025. URL: https://en.wikipedia.org/w/index.php?title=Period_(music)&oldid=1272601348 (visited on 03/26/2026).

[28] *Object-Oriented Programming*. In: *Wikipedia*. Mar. 23, 2026. URL: https://en.wikipedia.org/w/index.php?title=Object-oriented_programming&oldid=1344935717 (visited on 03/27/2026).

[29] Ivar Jacobson et al. *Object-Oriented Software Engineering*. Addison-Wesley, 1992. ISBN: 0-201-54435-0.

[30] F. Zurcher and Brian Randell. "Iterative Multi-Level Modeling - A Methodology for Computer System Design". In: Jan. 1, 1968, pp. 867–871.

[31] *AI | 2024 Stack Overflow Developer Survey*. URL: https://survey.stackoverflow.co/2024/ai/ (visited on 05/16/2026).

[32] Kyle Daigle Staff GitHub. *Survey: The AI Wave Continues to Grow on Software Development Teams*. The GitHub Blog. Aug. 20, 2024. URL: https://github.blog/news-insights/research/survey-ai-wave-grows/ (visited on 05/16/2026).

[33] *DORA | Accelerate State of DevOps Report 2024*. URL: https://dora.dev (visited on 05/16/2026).

[34] Zheyuan (Kevin) Cui et al. "The Effects of Generative AI on High-Skilled Work: Evidence from Three Field Experiments with Software Developers". In: (June 1, 2025). URL: https://www.microsoft.com/en-us/research/publication/the-effects-of-generative-ai-on-high-skilled-work-evidence-from-three-field-experiments-with-software-developers/ (visited on 05/16/2026).

[35] Inbal Shani Staff GitHub. *Survey Reveals AI's Impact on the Developer Experience*. The GitHub Blog. June 13, 2023. URL: https://github.blog/news-insights/research/survey-reveals-ais-impact-on-the-developer-experience/ (visited on 05/16/2026).

[36] *Codex | AI Coding Partner from OpenAI*. OpenAI. URL: https://openai.com/codex/ (visited on 05/16/2026).

[37] *Codex Is Now Generally Available*. OpenAI. May 15, 2026. URL: https://openai.com/index/codex-now-generally-available/ (visited on 05/16/2026).

[38] Ruotong Wang et al. "Investigating and Designing for Trust in AI-powered Code Generation Tools". In: FAccT 2024. May 17, 2023. URL: https://www.microsoft.com/en-us/research/publication/investigating-and-designing-for-trust-in-ai-powered-code-generation-tools/ (visited on 05/16/2026).

# APPENDICES

The appendices collect supporting material that would have interrupted the flow of the main text. They include a link to the project repository, a link to the online audio archive, and a broader set of non-cherry-picked examples from the third iteration. Together these appendices document the implementation, make the generated results easier to inspect and listen to, and provide additional material beyond what was necessary to include in the main chapters.

# A - GITHUB REPOSITORY

All code and latex-files used in this document are included in the Github repository linked below. Further explanations are given in the readme-file.

## Github repository link

- https://github.com/lima98/TTK4900 - Github repository

# B - AUDIO EXAMPLES

Audio-files for all the examples in the thesis can be found on my github.io page. Additionally we have a complete documentation of the final code available from the same page.

## Audio example archive

- https://lima98.github.io/TTK4900 - Audio example archive

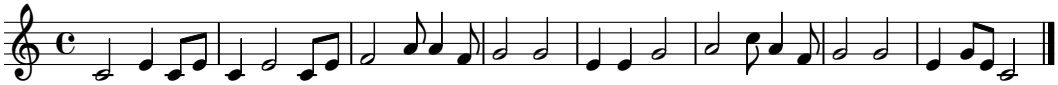
# C - THIRD ITERATION EXAMPLE APPENDIX

This appendix gathers a broader selection of outputs from the third iteration than could reasonably be included in the main discussion. These are non-cherry-picked examples, meaning I did not change the seed to try and find a better output, i simply generated them and then let them be.

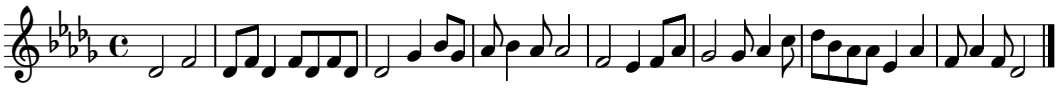
The examples are organized by purpose: first a survey across all twelve major keys, then a short comparison between parallel major and minor keys, followed by examples showing formal and modal control, rhythmic restriction, explicit harmonic control, longer spans, and several four-part textures.

## Automatic melodies in all twelve major keys

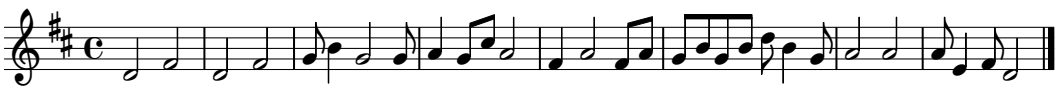
The following figures use the default third-iteration melody configuration with automatic harmony, eight bars, and the standard 4/4 setting. They are presented in the order C, D♭, D, E♭, E, F, F♯, G, A♭, A, B♭, and B. Together they show that the same generator can be applied across the full set of major keys while preserving the same overall workflow.



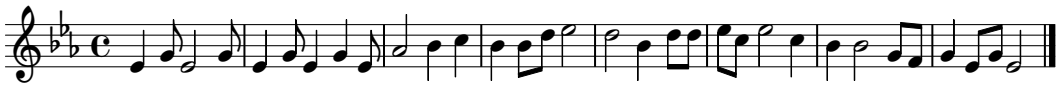
**Example C.1:** Automatically generated eight-bar melody in C major using the default automatic harmony plan and seed 4100. Listen in the online archive



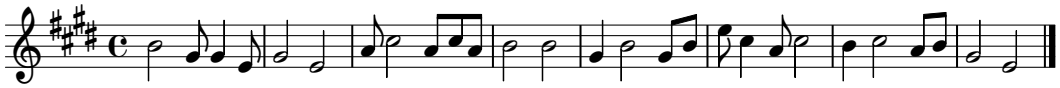
**Example C.2:** Automatically generated eight-bar melody in D♭ major using the default automatic harmony plan and seed 4101. Listen in the online archive



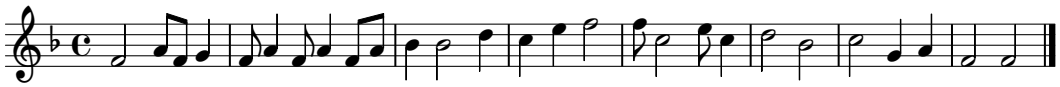
**Example C.3:** Automatically generated eight-bar melody in D major using the default automatic harmony plan and seed 4102. Listen in the online archive



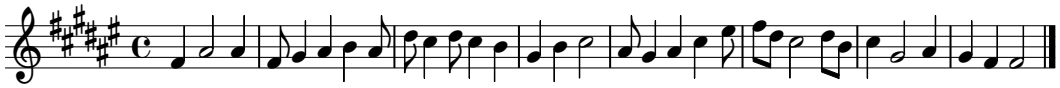
**Example C.4:** Automatically generated eight-bar melody in E♭ major using the default automatic harmony plan and seed 4103. Listen in the online archive



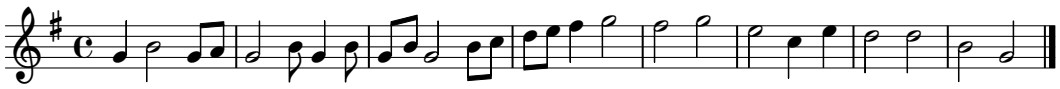
**Example C.5:** Automatically generated eight-bar melody in E major using the default automatic harmony plan and seed 4104. Listen in the online archive



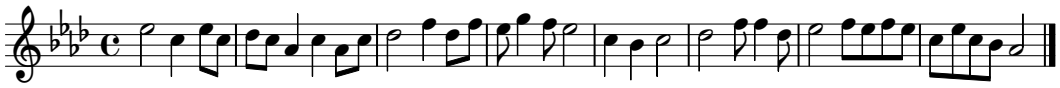
**Example C.6:** Automatically generated eight-bar melody in F major using the default automatic harmony plan and seed 4105. Listen in the online archive



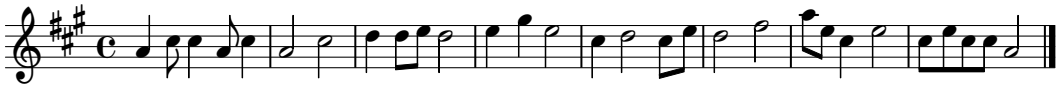
**Example C.7:** Automatically generated eight-bar melody in F♯ major using the default automatic harmony plan and seed 4106. Listen in the online archive



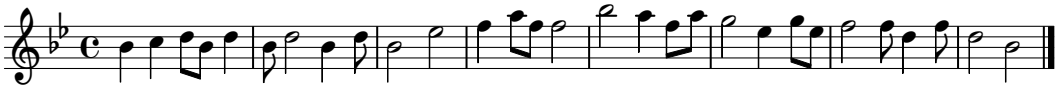
**Example C.8:** Automatically generated eight-bar melody in G major using the default automatic harmony plan and seed 4107. Listen in the online archive



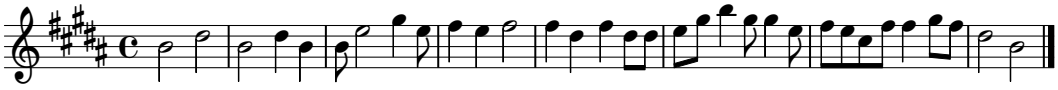
**Example C.9:** Automatically generated eight-bar melody in A♭ major using the default automatic harmony plan and seed 4108. Listen in the online archive



**Example C.10:** Automatically generated eight-bar melody in A major using the default automatic harmony plan and seed 4109. Listen in the online archive



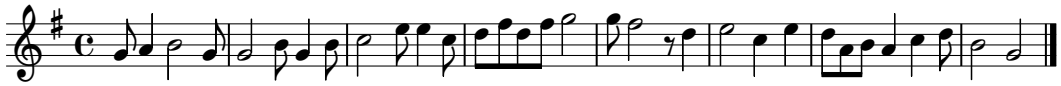
**Example C.11:** Automatically generated eight-bar melody in B♭ major using the default automatic harmony plan and seed 4110. Listen in the online archive



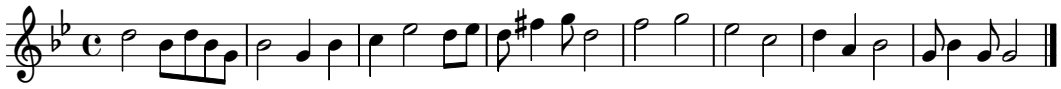
**Example C.12:** Automatically generated eight-bar melody in B major using the default automatic harmony plan and seed 4111. Listen in the online archive

## Parallel major and minor comparison

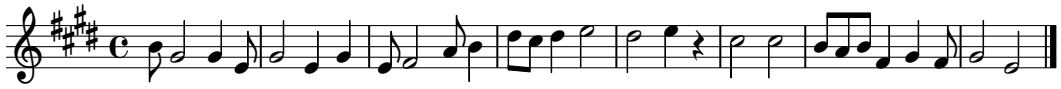
The major-key survey above shows that the generator can be used across the full chromatic set of tonics. The following four examples instead compare pairs of parallel keys directly. Each pair keeps the same tonic while changing only the mode, which makes it easier to see how the harmonic plan, pitch content, and general melodic colour change when the system moves from major to minor.



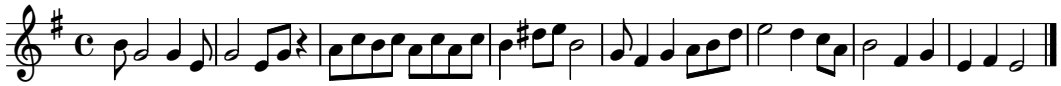
**Example C.13:** Automatically generated eight-bar melody in G major using the default automatic harmony plan and seed 4120. Listen in the online archive



**Example C.14:** Automatically generated eight-bar melody in G minor using the default automatic harmony plan and seed 4121. Listen in the online archive



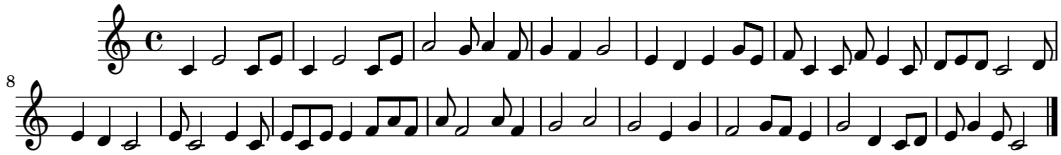
**Example C.15:** Automatically generated eight-bar melody in E major using the default automatic harmony plan and seed 4122. Listen in the online archive



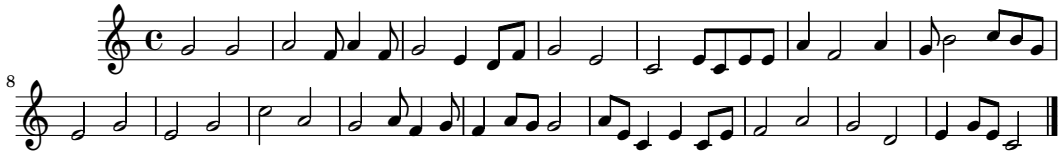
**Example C.16:** Automatically generated eight-bar melody in E minor using the default automatic harmony plan and seed 4123. Listen in the online archive

# Formal and modal variation

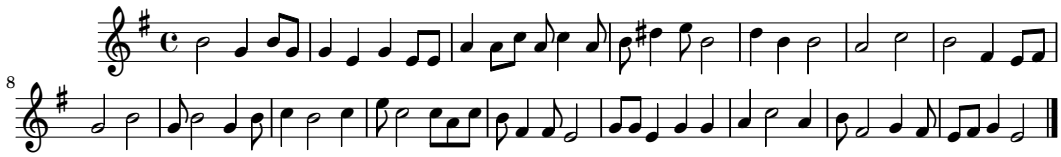
The next examples show that the same generator can be given more explicit musical structure. The first two use the sentence and period form plans discussed in the main text in major keys, while the last two show that the same formal ideas can also be applied in a minor mode.



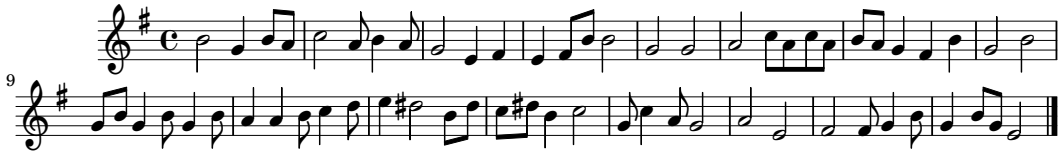
**Example C.17:** Sixteen-bar melody in C major using the explicit sentence form plan and seed 3100. Listen in the online archive



**Example C.18:** Sixteen-bar melody in C major using the explicit period form plan and seed 3200. Listen in the online archive



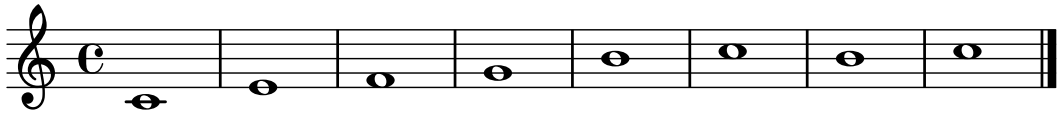
**Example C.19:** Sixteen-bar melody in E minor using the sentence form plan and seed 4301. This example is included to show the same structural ideas applied in a minor mode. Listen in the online archive



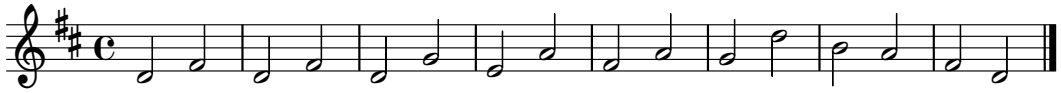
**Example C.20:** Sixteen-bar melody in E minor using the period form plan and seed 4302. This complements the earlier minor sentence example by showing the second large-scale form in the same mode. Listen in the online archive

## Rhythmic control

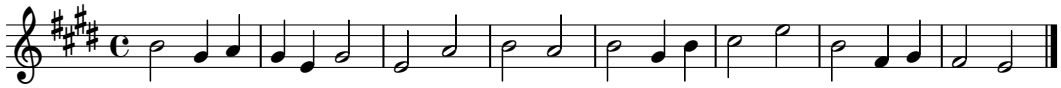
The generator can also be constrained through the allowed rhythmic values. The following examples use the same general workflow as the earlier melody outputs, but with different rhythm palettes. Together they show how the rhythmic surface becomes more static or more active depending on which durations are made available, ranging here from only whole notes to a continuous stream of eighth notes.



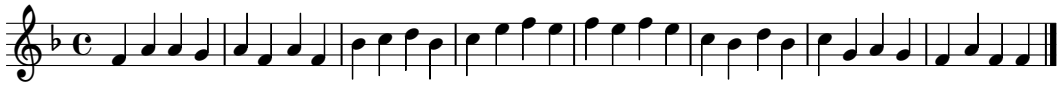
**Example C.21:** Automatically generated eight-bar melody in C major using only whole notes and seed 4410. Listen in the online archive



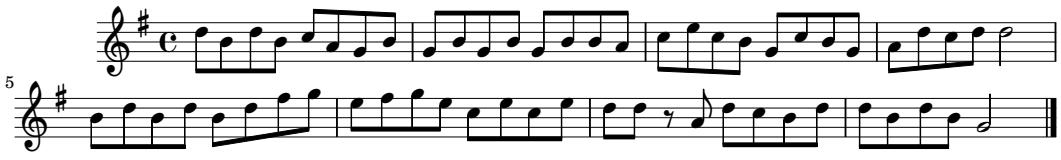
**Example C.22:** Automatically generated eight-bar melody in D major using only half notes and seed 4411. Listen in the online archive



**Example C.23:** Automatically generated eight-bar melody in E major using whole, half, and quarter notes with seed 4412. Listen in the online archive



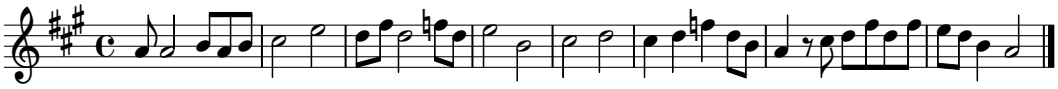
**Example C.24:** Automatically generated eight-bar melody in F major using only quarter notes and seed 4413. Listen in the online archive



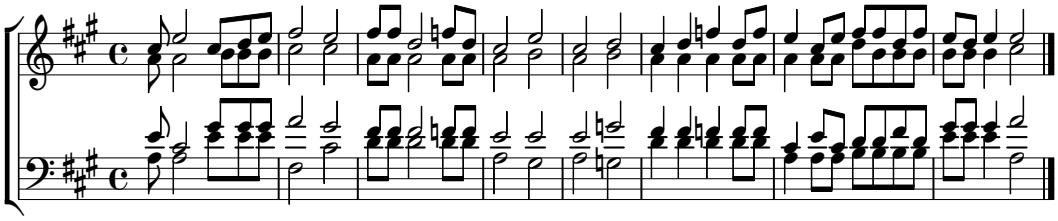
**Example C.25:** Automatically generated eight-bar melody in G major using only eighth notes and seed 4416. Listen in the online archive

## Explicit harmonic control

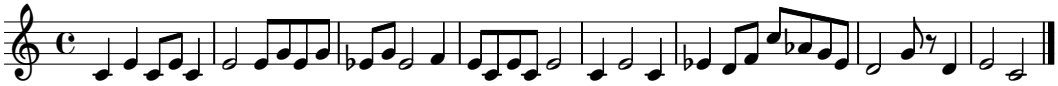
The third iteration also allows the harmonic progression to be specified directly from the command line. The first two examples use an explicit progression in A major that includes modal mixture through both the borrowed minor subdominant and a borrowed flat seventh chord. The third focuses on chromatic-mediant colour in C major, while the fourth is intentionally unstable and is included to show that explicit control is only useful when the harmonic choices themselves are musically convincing.



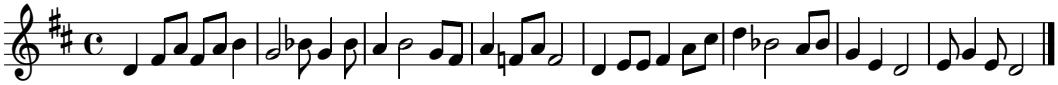
**Example C.26:** Melody generated in A major from an explicitly specified harmonic progression that includes the borrowed `iv` and `bVII` chords. Listen in the online archive



**Example C.27:** The same explicit A-major progression, including the borrowed `iv` and `bVII` chords, realized as a harmonized SATB-style texture. Listen in the online archive



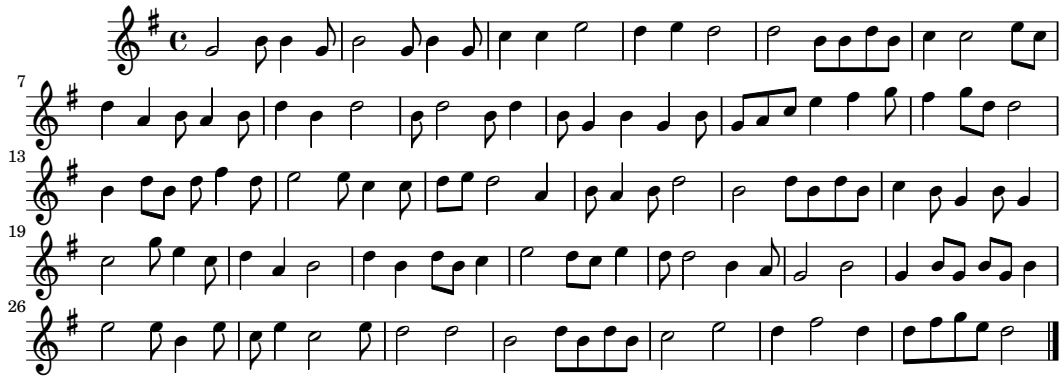
**Example C.28:** Melody generated in C major from an explicitly specified progression that emphasizes chromatic-mediant motion through `iii`, `bIII`, and `bVI`. Listen in the online archive



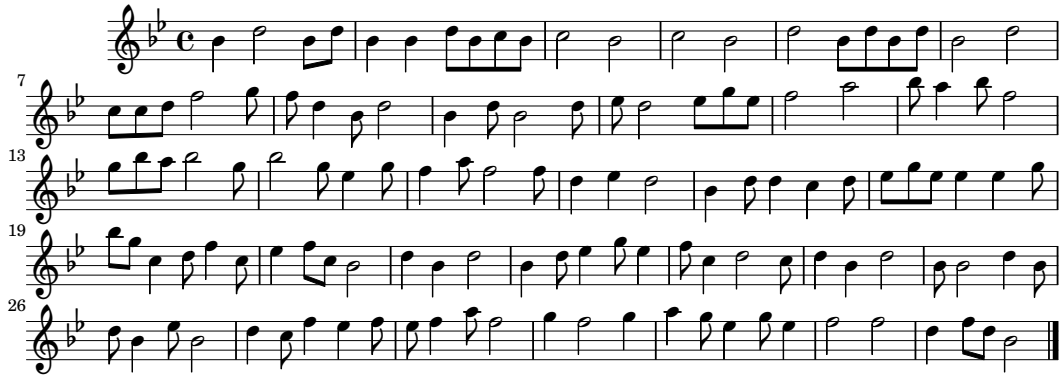
**Example C.29:** Melody generated in D major from an intentionally unstable harmonic plan. The example is included to show that direct harmonic control also makes it possible to steer the generator toward results that are structurally weaker if the chosen progression is not musically well-formed. Listen in the online archive

## Longer-span examples

The examples above are intentionally short and compact. The following two are substantially longer and are included to show how the third iteration behaves when it is allowed to unfold over a larger span. They are not complete songs in any formal sense, but they give a better impression of how the generator handles repetition, continuation, and cadential shaping over many bars.



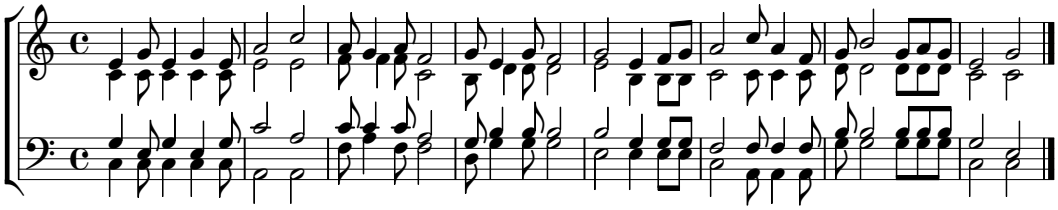
**Example C.30:** Thirty-two-bar melody in G major using the sentence form plan and seed 4200. Listen in the online archive



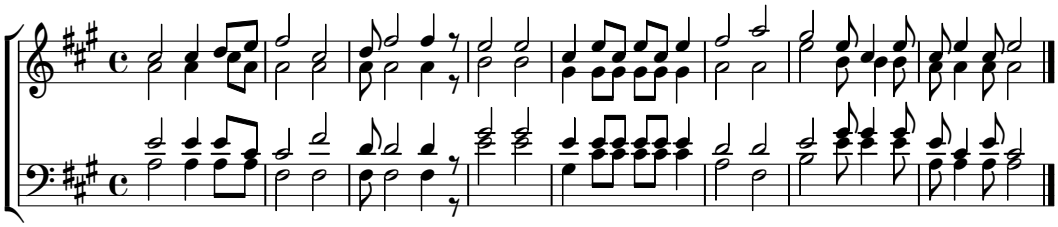
**Example C.31:** Thirty-two-bar melody in B♭ major using the period form plan and seed 4201. Listen in the online archive

# Automatic chorale output

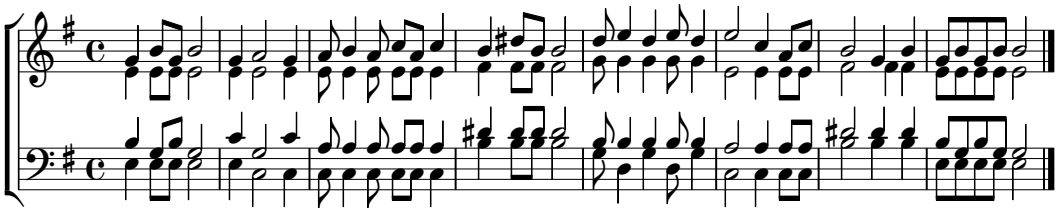
Finally, the appendix includes harmonized examples in several four-part textures. The first group contains shorter SATB-style results in both major and minor keys, included to show the basic extension from a generated melody to a four-part realization across different tonal settings. The next two examples demonstrate that the same harmonization approach can also be adapted to alternative ensemble layouts, here shown as TTBB and SSAA with suitable clefs and ranges. The last two examples are more directed: one is a longer SATB example with explicit modal mixture, while the final example keeps all melodic durations at the quarter-note level and uses a more baroque-inspired harmonic succession.



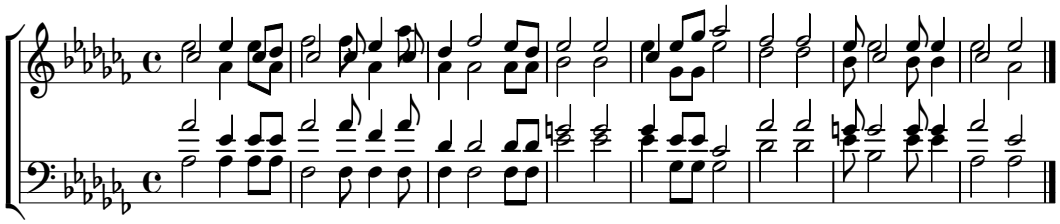
**Example C.32:** Automatically harmonized SATB-style example in C major using seed 2000. Listen in the online archive



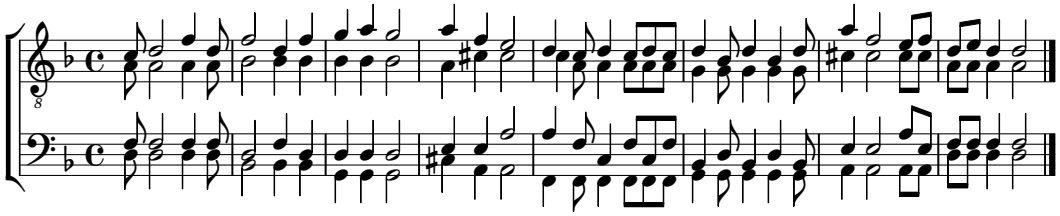
**Example C.33:** Automatically harmonized SATB-style example in A major using seed 4300. Listen in the online archive



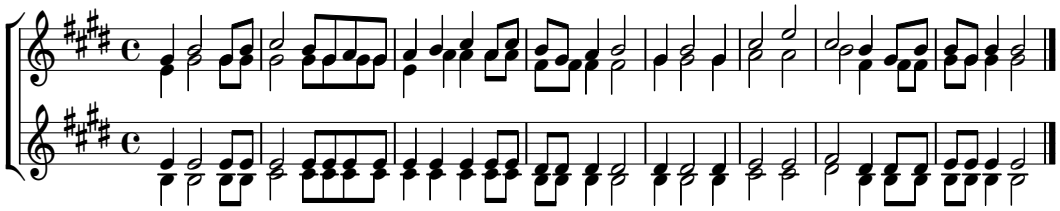
**Example C.34:** Automatically harmonized SATB-style example in E minor using seed 4401. Listen in the online archive



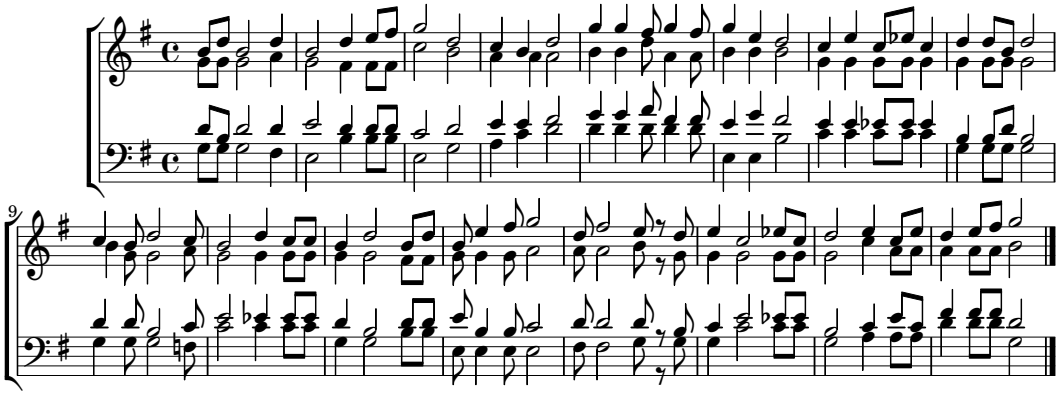
**Example C.35:** Automatically harmonized SATB-style example in A♭ minor using seed 4403. Listen in the online archive



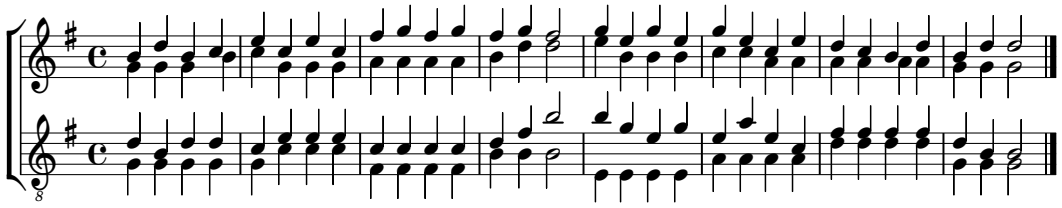
**Example C.36:** Automatically harmonized four-part example in D minor using a TTBB layout and seed 4414. Listen in the online archive



**Example C.37:** Automatically harmonized four-part example in E major using an SSAA layout and seed 4415. Listen in the online archive



**Example C.38:** Sixteen-bar harmonized SATB-style example in G major using an explicit harmonic progression with the borrowed `iv` and `bVII` chords, generated with seed 4402. Listen in the online archive



**Example C.39:** Eight-bar harmonized SATB-style example in G major using only quarter notes and an explicit progression inspired by baroque sequential harmony, generated with seed 4422. Listen in the online archive