TECHNISCHE HOCHSCHULE NÜRNBERG
GEORG SIMON OHM

Fakultät Informatik

HMM-based Musical Improvisation

Masterarbeit im Studiengang Informatik

vorgelegt von

Franziska Braun

Matrikelnummer 265 0920

Erstgutachter:	Prof. Dr. Korbinian Riedhammer
Zweitgutachter:	Prof. Dr. Sebastian Trump

Hinweis: Diese Erklärung ist in alle Exemplare der Abschlussarbeit fest einzubinden. (Keine Spiralbindung)

TECHNISCHE HOCHSCHULE NÜRNBERG
GEORG SIMON OHM

Prüfungsrechtliche Erklärung der/des Studierenden

Angaben des bzw. der Studierenden:

Name:	Braun	Vorname:	Franziska	Matrikel-Nr.:	2650920
Fakultät:	Informatik	Studiengang:	Informatik		
Semester:	Sommersemester	2021			

Titel der Abschlussarbeit:

HMM-based Musical Improvisation

Ich versichere, dass ich die Arbeit selbständig verfasst, nicht anderweitig für Prüfungszwecke vorgelegt, alle benutzten Quellen und Hilfsmittel angegeben sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

Nürnberg, 01.07.2021 F. Braun

Ort, Datum, Unterschrift Studierende/Studierender

Erklärung zur Veröffentlichung der vorstehend bezeichneten Abschlussarbeit

Die Entscheidung über die vollständige oder auszugsweise Veröffentlichung der Abschlussarbeit liegt grundsätzlich erst einmal allein in der Zuständigkeit der/des studentischen Verfasserin/Verfassers. Nach dem Urheberrechtsgesetz (UrhG) erwirbt die Verfasserin/der Verfasser einer Abschlussarbeit mit Anfertigung ihrer/seiner Arbeit das alleinige Urheberrecht und grundsätzlich auch die hieraus resultierenden Nutzungsrechte wie z.B. Erstveröffentlichung (§ 12 UrhG), Verbreitung (§ 17 UrhG), Vervielfältigung (§ 16 UrhG), Online-Nutzung usw., also alle Rechte, die die nicht-kommerzielle oder kommerzielle Verwertung betreffen.

Die Hochschule und deren Beschäftigte werden Abschlussarbeiten oder Teile davon nicht ohne Zustimmung der/des studentischen Verfasserin/Verfassers veröffentlichen, insbesondere nicht öffentlich zugänglich in die Bibliothek der Hochschule einstellen.

Hiermit ☒ genehmige ich, wenn und soweit keine entgegenstehenden Vereinbarungen mit Dritten getroffen worden sind,

☐ genehmige ich nicht,

dass die oben genannte Abschlussarbeit durch die Technische Hochschule Nürnberg Georg Simon Ohm, ggf. nach Ablauf einer mittels eines auf der Abschlussarbeit aufgebrachten Sperrvermerks kenntlich gemachten Sperrfrist

von 0 Jahren (0 - 5 Jahren ab Datum der Abgabe der Arbeit),

der Öffentlichkeit zugänglich gemacht wird. Im Falle der Genehmigung erfolgt diese unwiderruflich; hierzu wird der Abschlussarbeit ein Exemplar im digitalisierten PDF-Format auf einem Datenträger beigefügt. Bestimmungen der jeweils geltenden Studien- und Prüfungsordnung über Art und Umfang der im Rahmen der Arbeit abzugebenden Exemplare und Materialien werden hierdurch nicht berührt.

Nürnberg, 01.07.2021 F. Braun

Ort, Datum, Unterschrift Studierende/Studierender

Formular drucken

Kurzdarstellung

Diese Masterarbeit fand im Rahmen des Projekts Spirio Sessions statt, das sich zum Ziel gesetzt hat, Konzepte der freien Improvisation zwischen Mensch und Maschine in verschiedenen Forschungsrichtungen durch die Entwicklung von Prototypen, verschiedenen Kombinationen von Softwaremodulen und künstlerischer Evaluation zu erforschen. Der Schwerpunkt dieses Teilprojekts liegt auf der Untersuchung von automatisch generierten musikalischen Improvisationen unter Verwendung von Hidden Markov Modellen (HMMs). In diesem Rahmen wurde eine Anwendung für ein nahezu vollständig zugängliches und konfigurierbares HMM entwickelt, um musikalische Improvisationen mit der künstlichen Intelligenz (KI) zu ermöglichen. Anstatt auf eine feste HMM-Struktur und einen festen Musikgenerierungsprozess beschränkt zu sein, wie in bestehenden Ansätzen, bietet der vorgeschlagene Ansatz eine Auswahl verschiedener Konfigurationsparameter, die die Musikmodellierung und -generierung, und damit das finale musikalische Ergebnis, beeinflussen. Diese Parameter sind in eine Benutzeroberfläche eingebettet, die Teil einer entwickelten Client-Server-Anwendung ist. Um festzustellen, ob ko-kreative Prozesse zwischen KI und Mensch auf Augenhöhe möglich sind, wurde der entwickelte Ansatz zur Mensch-Maschine-Interaktion in den sogenannten Spirio-Sessions erprobt und evaluiert, in denen menschliche Musiker mit dem KI-gesteuerten Flügel “Spirio-R” interagierten.

Abstract

This master thesis took place within the Spirio Sessions project, which aims to explore concepts of free improvisation between humans and machines in different research directions through the development of prototypes, different combinations of software modules and artistic evaluation. The focus of this subproject is on the study of automatically generated musical improvisations using Hidden Markov Models (HMMs). Within this framework, an application for an almost fully accessible and configurable HMM was developed to enable musical improvisation with the AI. Instead of being limited to a fixed HMM structure and music generation process, as in existing approaches, the proposed approach provides a selection of different configuration parameters that influence music modeling and generation, and thus the final musical result. These parameters are embedded in a user interface that is part of a developed client-server application. To determine whether co-creative processes between AI and humans are possible at eye level, the developed approach to human-machine interaction was explored and evaluated in the so-called Spirio Sessions, in which human musicians interacted with the AI-controlled grand piano “Spirio-R”.

Contents

Chapter 1.

Introduction

Can artificial intelligence be creative? This is a fundamental question addressed by the Spirio Sessions project, in the context of which this Master's thesis took place. In order to answer this question, two further questions must be answered first: What is artificial intelligence? And what is creativity?

Motivation

The prospect of creating intelligent computers has fascinated many people for as long as computers have been around, but what does Artificial Intelligence mean, if even the term intelligence itself is difficult to define? The precise definition and meaning of the word intelligence, and even more so of Artificial Intelligence, is the subject of much discussion and has caused a lot of confusion [Kok 09]. One dictionary alone, for example, gives four definitions of Artificial Intelligence:

- An area of study in the field of computer science. Artificial intelligence is concerned with the development of computers able to engage in human-like thought processes such as learning, reasoning, and self-correction.
- The concept that machines can be improved to assume some capabilities normally thought to be like human intelligence such as learning, adapting, selfcorrection, etc.
- The extension of human intelligence through the use of computers, as in times past physical power was extended through the use of mechanical tools.
- In a restricted sense, the study of techniques to use computers more effectively by improved programming techniques.

  [Smit 03]

Instead of looking at a general definition of Artificial Intelligence, it is easier to limit to the definition of artificially intelligent systems. There are many definitions around, but most of them can be classified into the following four categories [Brin 20]:

- systems that think like humans
- systems that act like humans
- systems that think rationally
- systems that act rationally

So if, according to this definition of AI, a machine can think and act like a human, doesn't that also mean it must be creative? Creativity is a trait normally only attributed to living beings, but in the context of artificial intelligence, it is interesting whether the term can be extended to this area. According to the first definition found in Webster's Dictionary [Smit 03], being creative is marked by the ability or power to create. By this definition, AI systems could be said to be creative, since they can generate new output from learned input. But a second definition specifies it to having the quality of something created rather than imitated. This is where most AI systems reach their limits. While they can create new things, AIs are usually imitating from learned information. The extent of their creativity is determined and limited by the input they get. This shows that another aspect is important in defining the creativity process: inventiveness. Creativity also means being imaginative and having original ideas. An AI cannot produce more than it has learned, and things an AI has never seen it cannot produce.
This statement may be true at the moment for most AI systems that use neural networks, for example. But the Spirio Sessions project aims to determine whether joint creative processes between AI and humans are possible at eye level. A radio report from SWR2 [Schl 21] presented the project alongside other existing approaches that pursue music generation with artificial intelligence. In this report it becomes clear that the approach differs from other existing approaches to deal with existing music. Instead of making style copies, as music theorists would say, it aims first completely into free improvisation.
If the AI is to imitate music styles, the deep learning approach is chosen. In the Spirio prototypes, however, training is done with other learning approaches, which require fewer learning processes, but develop special degrees of freedom: Markov chains and Hidden Markov Models. These are statistical methods that can be trained much faster, easier and less data intensive. The principle of Markov chains is that a computer processes a sequence of values using methods from stochastics, i.e. probability theory and statistics, and tries to calculate what the next value could be. In the case of music this sequence could be a sequence of notes, for example, in which the AI constantly recalculates what the next note should sound. Thus it comes to the fact that the AI improvises so to speak. Technically, it is possible to experiment with the degrees of attention of this artificial

intelligence by defining how much it listens and how much it outputs. Artistically, this results in very different paths in these free improvisations. During the joint interaction of a musician with the system, it is constantly changing who is giving something. On the one hand, the system listens, learns what is being played and reacts to it; on the other hand, the improviser hears what the system is playing and can in turn react to it. This creates a multi-layered interplay in which it is often not really possible to say who originally initiated an idea. According to most scientists, this is where AI really gets creative [Schl 21].

Objective

Building on the statistical approach of the project's first prototype, which uses Markov chains for music generation, the focus of this subproject is to investigate automatically generated musical improvisation using Hidden Markov Models (HMM). The goal is to use these models to generate a rhythmic improvisation to a recorded input melody so that the generated output can be seen as an interaction or reaction to it.
Since the system is primarily applied and tested in the context of Spirio Sessions, one requirement is live adaptation to the input of the improvisation partner. This is achieved in HMM through continuous learning of different musical elements. Both the learning process and the music generation process must occur in real time and latency must be kept to a minimum, which is already inherent in the architecture of HMMs.
In order to determine which external and internal factors influence the improvisation and in what way, different parameters were introduced that affect the final melodic result. These parameters are configurable through the frontend of a client-server architecture and affect the modeling and generation of music in the backend. This results in a fully accessible and configurable prototype for musical interaction between an AI (the HMM) and a human musician. The parameters examine the influence of the input, the output, and especially the HMM topology and training.
One parameter for the input is the texture of the played instrument, that can be monophonic (e.g. saxophone) or polyphonic (e.g. piano). For the output, different sampling approaches and a degree of randomness can be tested. The parameters for the HMM can be further divided into parameters that affect the HMM topology and parameters that affect the HMM training.
The HMM topology is defined by the set of states, the arrangement of state transitions including their probabilities, the observation sequence, the observation likelihoods, and the initial probability distribution over the states [Jura 09]. For musical improvisation, the states and observations can be assigned different meanings [Maro 97], such as notes, note durations, velocities, intervals, or chords [Simo 08].
In HMM training, a distinction can be made between whether training is event-triggered,

e.g., after a note is played, or time-triggered, e.g., after each quarter beat. In addition to triggering, parameters that affect the training process itself are also examined, such as the window size, which indicates how many past observations are included in the retraining, the weighting parameter, which indicates how heavily the retrained matrices are weighted, and the probabilities for the emissions and transitions that can be pre-trained on MIDI data or initialized with certain distributions such as Gaussian, Discrete, or Random. It also makes a difference if training is performed with a flat start or if retraining algorithms like expectation–maximization (EM) are applied [Jura 09].
For melody generation, it is possible to draw samples from the HMM or make a prediction based on an observed sequence. The number of samples generated and the sample rate then affect the resulting melody. To perform the experiments, all parameters were embedded as configurable units in a user interface so that they can be used by musicians, as well as technicians, to test the influence on the generated improvisations.

Spirio Sessions

In collaboration with the Technical University of Nuremberg and the University of Music, the Spirio Session [Trum 21] project was founded in 2020, which is funded by LEONARDO - Center for Creativity and Innovation. It is an ongoing interdisciplinary research project that aims to explore concepts of free improvisation between humans and machines in different research directions through the development of prototypes, different combinations of software modules and artistic evaluation.
Approaches to human-machine interaction are explored in different settings around the digital self-playing grand piano “Spiro-R” by Steinway & Sons. For this purpose, the student project participants develop technical concepts that are continuously evaluated in artistic applications, the so-called “Spirio Sessions”.
The goal is to explore methods to co-creatively collaborate with an AI player embodied in the Spirio grand piano as an equal improvisation partner. The incentive to use a grand piano instead of loudspeakers for sonic realization is to give the computer-generated musical material a physical presence in this human-machine collaboration scenario. This allows the framework of a realistic duo setting, consisting of the AI-controlled grand piano and a human musician, in which the various computer-based approaches to interactive generation of musical material are explored.

Chapter 2.

Data

Table 2.1.: Detailed list of recorded piano and saxophone data.

Name	BPM	Length	mon/pol	Description
Piano1	100		mon	Pentatonic impro in one key
Piano2	100		mon	Pentatonic impro in different keys
Piano3	150		mon	Bass line blues form
Piano4	150		pol	Comping/Accompanying with chords blues form
Piano5	120		mon	Blues impro single line
Piano6	120		pol	Blues impro with chords
Piano7	100		mon	Arpeggios with pedal
Piano8	160		mon	Impro dorian-mode multiple keys
Piano9	180		mon	Donna Lee theme single line
Sax1	68		mon	Blues style
Sax2	-		mon	Multiphonics
Sax3	112		mon	Funky
Sax4	-		mon	Glissandi, Ballade
Sax5	56		mon	Slow blues
Sax6	156		mon	Virtuose melody
Sax7	116		mon	Funk solo
Sax8	102		mon	Calypso
Sax9	-		mon	Octave leaps
Sax10	88		mon	Bass line

Compared to classical machine learning approaches with neural networks, this approach can use relatively little data to train the Hidden Markov Model. The data for the application scenario in Spirio Sessions was self recorded by musicians of the project and the HMM already achieved convincing results in this framework with little training data. Without pre-training, the HMM even adapts more closely to the player input, since its structure is merely built on it. In general, the data can be classified into pre-training data and live training data. They are both in MIDI format (cf. 3.4), which has the advantage that the musical information is in symbolic form and is thus easier to process and less error-prone than e.g. the processing of pitch information.
For the pre-training, midi files of saxophone and piano were recorded beforehand. This

resulted in 10 saxophone and 9 piano files that can be classified in the genre of jazz, which is the featured genre in Spirio Sessions. Table 2.1 lists the recorded data that varies in musical texture, monophonic and polyphonic, as well as in tempo, measured in BPM, and in recording length, measured in seconds. The data also covers various musical stylistic devices such as pentatonic, bass line, comping, blues improvisation, arpeggios, Dorian key, funk and many more. The variety in the data is intended to test how adaptable the HMM is in the above categories.
During recording, it can sometimes happen that invalid MIDI is generated. Therefore, in a pre-processing step, all notes whose end time is before their start time, as well as duplicate NOTE OFF messages, have been removed.
In the end, the Jazz ML ready MIDI dataset [Sai 18] was chosen to train the HMM on a larger dataset as well. This dataset consists of over 935 jazz music tracks in MIDI format downloaded from free MIDI websites like `https://freemidi.org/`. It was created to generate custom music using deep learning. Therefore, the actual dataset is a CSV file containing the name of the music pieces and the notes extracted from the MIDI files, as well as some other related features. However, in this approach, the MIDI files are used directly and the required information is extracted by itself. The AI should in the best case learn how a jazz pianist plays the piano, but the music pieces of Jazz ML ready MIDI dataset contain a wide variety of instruments. For this reason, the piano tracks were extracted from the data during pre-processing. In MIDI, the instrument sound or "program" for each of the 16 possible MIDI channels is selected with the Program Change command, which has a program number parameter [The 21]. In general MIDI alone, there are 128 program numbers, with values usually given in the range 1 to 128. The piano range is from 1 to 8, with their assignment as shown in Table 2.2.

Table 2.2.: MIDI program numbers for piano instrument family.

1	**2**	**3**	**4**
Acoustic Grand Piano	Bright Acoustic Piano	Electric Grand Piano	Honky-tonk Piano
5	**6**	**7**	**8**
Electric Piano 1	Electric Piano 2	Harpsichord	Clavinet

This approach ignores numbers 7 and 8, since harpsichord and clavinet are very specific types of piano that were found to be unsuitable for mapping to the grand piano. Accordingly, only the tracks with a program number from 1 to 6 were filtered out of all MIDI files. Moreover, to map the MIDI pitches to the standard 88-key claviature, only the pitches in the MIDI number range from 21 to 108 were extracted.
Descriptions of the live training data generated in the Spirio Sessions experiments are listed in Appendix A.3 and are discussed in more detail in Chapter 5. They often involve free tonal material or an abrupt change from one musical playing style to another to test the adaptability of the HMM in a live scenario.

Chapter 3.

Method

Hidden Markov Models (HMMs) are well known for their applications within numerous fields, especially in speech recognition [Rabi 86] [Schu 95], but also in thermodynamics, physics [Bech 15], chemistry [Liu 03], economics, finance [Bhar 04], signal processing [Vase 01], and pattern recognition [Star 95], to name a few. In music, they are already used for pitch tracking [Bene 11] and a few approaches for music generation (cf. 3.5) exist as well.

This approach resorts to a Hidden Markov Model for generation of musical improvisations. Since the HMM is based on and extends the principle of Markov chains, this chapter first introduces Markov chains and then builds on them to explain the principle of HMMs. The findings of Jurafsky, Martin [Jura 09] and Rabiner [Rabi 86] form the basis for these two sections (3.1 and 3.2).

Afterwards, the musical dimensions based on music theory are introduced. In order to model these musical dimensions in an HMM, ways of representation with the chosen MIDI format are then presented. Based on this, different approaches to modeling and generating music are shown in the related work section.

Finally, the approach proposed in this thesis is explained in detail, taking elements from related works and combining them in an application almost fully accessible, configurable HMM.

3.1. Markov Chain

A Markov chain is a model that can make statements about the probabilities of sequences of random variables. These random variables are called states, each of which can take values from a defined set. These sets can be words, or tags, as in the speech recognition application, or symbols, which can represent anything, such as the weather or, in this use case, notes. A Markov chain assumes that only the current state is important for predicting a sequence in the future, and that past states have no influence on it except that they led to the current state. To use Jurafsky and Martin's [Jura 09] comparison, this is like saying that to predict tomorrow's weather, only today's weather could be examined, but yesterday's weather should not be looked at.

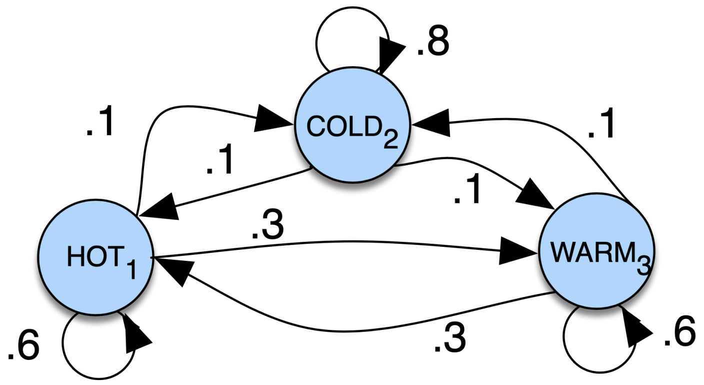


Figure 3.1.: A Markov chain for weather from [Jura 09], showing states and transitions. A start distribution π is required; setting $\pi = [0.1,\ 0.7,\ 0.2]$ would mean a probability 0.7 of starting in state 2 (cold), probability 0.1 of starting in state 1 (hot), etc.

Considering a sequence of state variables $q1, q2, ..., qi$, a Markov model represents the Markov assumption about the probabilities of this sequence: that the probability of getting into an arbitrary state only depends upon the current state, but not on the previous states.

$$\textbf{Markov Assumption: } P(q_i = a|q_1...q_{i-1}) = P(q_i = a|q_{i-1}) \tag{3.1}$$

A stochastic process which fulfils the Markov Assumption is called a first order Markov chain. An nth-order Markov chain is a stochastic process that satisfies the following condition:

$$P(q_i = a|q_1...q_{i-1}) = P(q_i = a|q_{i-1}...q_{i-n}) \tag{3.2}$$

The probability of reaching the next state in this process depends on the n previous states. That means when predicting the future the past n states and the present state are considered. Generally, the term Markov chain is used as a synonym for a first-order Markov Chain.

Figure 3.1 shows a Markov chain to predict the probability for a sequence of weather events described by the vocabulary consisting of HOT, COLD, and WARM. Here, the states form the nodes of the graph, which are connected by edges representing the transitions between the states. Transitions are annotated with probabilities, which indicate the chance that a certain state change might occur. The probabilities of leaving a particular state must sum to 1. Based on the models in Figure 3.1, each sequence from the vocabulary can be assigned a probability.

A Markov chain is formally specified by the following components:

$Q = q_1 q_2 ... q_N$	a set of N states
$A = a_{11} a_{12} ... a_{n1} ... a_{nn}$	a transition probability matrix A, each a_{ij} representing the probability of moving from state i to state j, s.t. $\sum_{j=1}^{n} a_{ij} = 1 \; \forall i$
$\pi = \pi_1, \pi_2, ..., \pi_N$	an initial probability distribution over states. π_i is the probability that the Markov chain will start in state i. Some states j may have $\pi_j = 0$, meaning that they cannot be initial states. Also, $\sum_{i=1}^{n} \pi_i = 1$

3.2. Hidden Markov Model

The extension of the Markov chain is the Hidden Markov Model, which is usually defined as a stochastic finite state machine [Kohl 07]. Unlike regular Markov chains, the states Q are hidden, i.e., they cannot be observed directly. As with Markov chains, the transition probabilities A indicate the probability of state transitions occurring. These probabilities, as well as the initial state probabilities π, are discrete. Each state has a set of possible emissions V and a discrete or continuous probability distribution B for those emissions. Unlike states, emissions can be observed and thus provide some information about the hidden states, such as the most likely underlying hidden state sequence that led to a particular observation O. This is called the decoding problem which, together with the evaluation problem and the learning problem, is one of the three main problems formulated for HMMs.

An HMM is formally specified by the following components:

$Q = q_1 q_2 ... q_N$	a set of N states
$A = a_{11} ... a_{ij} ... a_{NN}$	a transition probability matrix A, each a_{ij} representing the probability of moving from state i to state j, s.t. $\sum_{j=1}^{N} a_{ij} = 1 \; \forall i$
$O = o_1 o_2 ... o_T$	a sequence of T observations, each one drawn from a vocabulary $V = v_1, v_2, ..., v_V$
$B = b_i(o_t)$	a sequence of observation likelihoods, also called emission probabilities, each expressing the probability of an observation o_t being generated from a state i

$\pi = \pi_1, \pi_2, ..., \pi_N$	an initial probability distribution over states. π_i is the probability that the Markov chain will start in state i. Some states j may have $\pi_j = 0$, meaning that they cannot be initial states. Also, $\sum_{i=1}^{n} \pi_i = 1$

A first-order HMM fulfills the first-order Markov assumption. Moreover, it instantiates a further assumption that the probability of an initial observation o_i depends only on the state that generated the observation q_i, and not on other states or other observations:

$$\textbf{Output Independence: } P(o_i|q_1...q_i, ..., q_T, o_1, ..., o_i, ..., o_T) = P(o_i|q_i) \tag{3.3}$$

An influential tutorial by Rabiner (1986) [Rabi 86] introduced the idea that hidden Markov models should be characterized by three fundamental problems:

- **Likelihood Problem:**
  Given an observation sequence O and an HMM with complete model parameters with complete model parameters, meaning that transition and emission probabilities are known, $\lambda = (A, B)$, determine the likelihood of O to be observed under the given model $P(O|\lambda)$.

- **Decoding Problem:**
  Given an observation sequence O and an HMM with complete model parameters $\lambda = (A, B)$, this problem asks for the most probable underlying sequence Q of hidden states that led to this particular observation.

- **Learning Problem:**
  Given an observation sequence O and the elemental structure of the HMM, meaning only the set of states is known, learn the model parameters A and B.
  In other words: the parameters of the HMM have to be trained.

To solve the fundamental problems there are three main algorithms, which are introduced in the following subsections.

3.2.1. Forward Algorithm

The forward algorithm is a quite efficient solution to the likelihood problem, which is to compute the likelihood of a particular observation sequence. For a Markov chain where the surface observations and the hidden states coincide, the probability of an observation sequence can be calculated by following the states labeled with that sequence and multiplying the probabilities along the edges. For a Hidden Markov Model this is not so simple, since the probability of an observation sequence is to be determined without knowing the underlying hidden state sequence. Assuming the hidden state sequence is given, the output probability of an observation sequence could be easily calculated as follows.

Since each hidden state generates only a single observation, the sequence of hidden states and the sequence of observations have the same length. [1] Given this one-to-one mapping and the Markov assumptions, for a given sequence of hidden states $Q = q_0, q_1, q_2, ..., q_T$ and an observation sequence $O = o_1, o_2, ..., o_T$ the likelihood of the observation sequence is:

$$P(O|Q) = \prod_{i=1}^{T} P(o_i|q_i) \tag{3.4}$$

However, since the hidden state sequence is unknown the probability of the observation sequence must instead be calculated by summing over all possible state sequences, weighted by their probability. To do this, the calculation of the joint probability of being in a particular state sequence Q and generating a particular sequence O of events is first introduced:

$$P(O, Q) = P(O|Q) \times P(Q) = \prod_{i=1}^{T} P(o_i|q_i) \times \prod_{i=1}^{T} P(q_i|q_{i-1}) \tag{3.5}$$

From this, the total probability of the observations is obtained by summing over all possible hidden state sequences as follows:

$$P(O) = \sum_{Q} P(O, Q) = \sum_{Q} P(O|Q)P(Q) \tag{3.6}$$

For an HMM with N hidden states and an observation sequence of T observations, this means that there are N^T possible hidden sequences. In real applications, N^T becomes extremely large, so that it would be too computationally expensive to calculate the total observation probability in this way.

Therefore, instead of this highly exponential algorithm, the more efficient forward algorithm with $O(N^2T)$ is used. This algorithm works like a dynamic programming algorithm, using a table to store intermediate values as it computes the probability of the observation sequence. The computation is still done by summing the probabilities of all possible hidden state paths that could generate the observation sequence, but in an efficient way, since each of these paths is implicitly folded into a single forward trellis. In this trellis, each cell $\alpha_t(j)$ represents the probability of being in state j after obtaining the first t observations, given the model λ.

$$\alpha_t(j) = P(o_1, o_2 ... o_t, q_t = j|\lambda) \tag{3.7}$$

[1] In a variant of HMMs called segmental HMMs (in speech recognition) or semi-HMMs (in text processing) this one-to-one mapping between the length of the hidden state sequence and the length of the observation sequence does not hold.

Here $q_t = j$ means that the t-th state in the state sequence is state j. The probability $\alpha_t(j)$ can be calculated by summing over the extensions of all paths leading to the current cell. Given a state q_j at time t, then $\alpha_t(j)$ is calculated as follows:

$$\alpha_t(j) = \sum_{i=1}^{N} \alpha_{t-1}(i) a_{ij} b_j(o_t) \tag{3.8}$$

Where $\alpha_{t-1}(j)$ is the forward path probability from the previous time step, a_{ij} is the transition probability from previous state q_i to current state q_j and $b_j(o_t)$ is the state observation likelihood of the observation symbol o_t given the current state j.
Two formal definitions of the forward algorithm can be found in the Python pseudocode in Listing 3.1 and in a statement of the recursion after it.

```
def forward_algo(observations of len T, state-graph of len N):
    create a probability matrix forward[N,T]
    for state s in range(1, N):            #initialization step
        forward[s,1] = $\pi_s * b_s(o_1)$
    for time_step t in range(2, T):        #recursion step
        for state s in range(1, N):

            forward[s,t] = $\sum_{s'=1}^{N} forward[s', t-1] * a_{s',s} * b_s(o_t)$

    forwardprob = $\sum_{s=1}^{N} forward[s,T]$          #termination step
    return forwardprob
```

Listing 3.1: Python pseudocode for the forward algorithm, where forward[s,t] represents $\alpha_t(s)$.

Initialization:

$$\alpha_1(j) = \pi_j b_j(o_1); \quad 1 \le j \le N$$

Recursion:

$$\alpha_t(j) = \sum_{i=1}^{N} \alpha_{t-1}(i) a_{ij} b_j(o_t); \quad 1 \le j \le N, 1 < t \le T$$

Termination:

$$P(O|\lambda) = \sum_{i=1}^{N} \alpha_T(i)$$

3.2.2. Viterbi Algorithm

The decoding problem is solved by the decoding algorithm, also called the Viterbi algorithm. The task of the decoder is to determine which sequence of hidden variables is the underlying source of a sequence of observations.
Theoretically, the best sequence could be found by running the forward algorithm for each possible hidden state sequence and computing the probability of the observation sequence given that hidden state sequence. This would then allow to select the hidden state sequence with the maximum observation probability. However, another problem occurring here is also that the computational cost would be far too large since there is an exponentially large number of state sequences.
The Viterbi algorithm, like the forward algorithm, is a type of dynamic programming that uses a dynamic trellis to solve this task. The idea is to process the sequence of observations from left to right, filling in the grid. Each grid cell, $v_t(j)$, represents the probability that the given HMM λ is in state j after seeing the first t observations and passing through the most likely state sequence $q_1, ..., q_{t-1}$. Viterbi fills each cell recursively by taking the most likely path that could lead to that cell. The probability expressed by each cell $v_t(j)$ is consequently:

$$v_t(j) = \max_{q_1,\cdots,q_{t-1}} P(q_1...q_{t-1}, o_1, o_2...o_t, q_t = j|\lambda) \tag{3.9}$$

The most likely path is represented by the maximum over all possible previous state sequences. Once the probability of being in each state at time $t-1$ has been calculated, the Viterbi probability is calculated by taking the most likely of the extensions of the paths leading to the current cell. Given a state q_j at time t, then $v_t(j)$ is calculated as follows:

$$v_t(j) = \max_{i=1}^{N} v_{t-1}(i) a_{ij} b_j(o_t) \tag{3.10}$$

Where $v_{t-1}(i)$ is the Viterbi path probability from the previous time step, a_{ij} is the transition probability from previous state q_i to current state q_j and $b_j(o_t)$ is the state observation likelihood of the observation symbol o_t given the current state j.

The Listing 3.2 shows the Python pseudocode for the Viterbi algorithm. It can be seen that the Viterbi algorithm is almost identical to the forward algorithm, except that it takes the maximum over the previous path probabilities, while the forward algorithm takes the sum. Also, unlike the forward algorithm, the Viterbi algorithm has an additional backpointer component. This is because the Viterbi algorithm must generate a probability and also the most likely sequence of states. To obtain this sequence of states, the path of hidden states that led to each state is tracked and at the end the best path is traced back to the beginning (Viterbi backtrace).

Finally, a formal definition of Viterbi recursion is as follows:

Initialization:

$$v_1(j) = \pi_j b_j(o_1); \quad 1 \le j \le N$$

$$bt_1(j) = 0; \qquad 1 \le j \le N$$

Recursion:

$$v_t(j) = \max_{i=1}^{N} v_{t-1}(i) a_{ij} b_j(o_t); \quad 1 \le j \le N, 1 < t \le T$$

$$bt_t(j) = \arg\max_{i=1}^{N} v_{t-1}(i) a_{ij} b_j(o_t); \quad 1 \le j \le N, 1 < t \le T$$

Termination:

$$\text{The best score:} \quad P* = \max_{i=1}^{N} v_T(i)$$

$$\text{The start of backtrace:} \quad q_T* = \arg\max_{i=1}^{N} v_T(i)$$

```
def viterbi_algo(observations of len T, state-graph of len N):
    create a path probability matrix viterbi[N,T]
    for state s in range(1, N):          #initialization step
        viterbi[s,1] = $\pi_s * b_s(o_1)$
        backpointer[s,1] = 0
    for time_step t in range(2, T):      #recursion step
        for state s in range(1, N):

            viterbi[s,t] = $\max_{s'-1}^{N} viterbi[s', t-1] * a_{s',s} * b_s(o_t)$

            backpointer[s,t] = $\arg\max_{s'-1}^{N} viterbi[s', t-1] * a_{s',s} * b_s(o_t)$

    bestpathprob = $\max_{s=1}^{N} viterbi[s, T]$          #termination step
    bestpathpointer = $\arg\max_{s=1}^{N} viterbi[s, T]$   #termination step
    bestpath = the path starting at state bestpathpointer ,
                  that follows backpointer[] to states back in time
    return bestpath , bestpathprob
```

Listing 3.2: Python pseudocode for the viterbi algorithm, where viterbi[s,t] represents $v_t(s)$.

3.2.3. Forward-Backward Algorithm

The standard algorithm for HMM training is the forward-backward algorithm, which solves the learning problem for HMMs, where the parameters A and B of an HMM are to be learned based on a given observation sequence. This learning algorithm is also known as the Baum-Welch algorithm [Baum 72], which is a special case of the expectation-maximization or EM algorithm [Demp 77]. This algorithm can be used to train both the transition probabilities A and the emission probabilities B of the HMM. EM is an iterative algorithm that computes an estimate for the probabilities at the beginning, which it then uses to compute a better estimate, and so on, iteratively improving the probabilities it learns. In the same way, the Baum-Welch algorithm estimates the transition and emission probabilities. To do this, it calculates the forward probability for an observation and then divides this probability mass among all the different paths that contributed to the forward probability. The Baum-Welch algorithm also defines another probability related to the forward probability, the backward probability β, which is the probability of seeing the observations from time $t+1$ to the end, given the condition of being in state i at time t and given the model λ.

$$\beta_t(i) = P(o_{t+1}, o_{t+2}...o_T | q_t = i, \lambda) \tag{3.11}$$

The backward probability β is calculated inductively in a similar way to the forward algorithm.

Initialization:

$$\beta_T(i) = 1; \quad 1 \le i \le N$$

Recursion:

$$\beta_t(i) = \sum_{j=1}^{N} a_{ij} b_j(o_{t+1}) \beta_{t+1}(j); \quad 1 \le i \le N, 1 \le t < T$$

Termination:

$$P(O|\lambda) = \sum_{j=1}^{N} \pi_j b_j(o_1) \beta_1(j)$$

Together, the forward and backward probabilities are used to compute the transition probability a_{ij} and the observation probability $b_i(o_t)$ from an observation sequence, although the actual path taken through the model is hidden. First, $\widehat{a}_{ij}$ can be estimated by a variant of simple maximum likelihood estimation:

$$\widehat{a}_{ij} = \frac{\text{expected number of transitions from state } i \text{ to state } j}{\text{expected number of transitions from state } i} \tag{3.12}$$

For the numerator, the total number for transition $i \rightarrow j$ is estimated. This requires estimating the probability that a given transition $i \rightarrow j$ occurred at a particular time t

in the observation sequence. If this probability were known for each particular time t, the sum over all time points t could be taken to determine the numerator. The probability ξ_t of being in state i at time t and in state j at time $t+1$, given the observation sequence and the model, is defined as:

$$\xi_t(i,j) = P(q_t = i, q_{t+1} = j|O,\lambda) \tag{3.13}$$

To compute ξ_t, a similar probability is first calculated, which differs in that it includes the probability of the observation, which is evident in the different conditioning of O from equation 3.13:

$$\text{not-quite-}\xi_t(i,j) = P(q_t = i, q_{t+1} = j, O|\lambda) \tag{3.14}$$

Four probabilities are included in the calculation of not-quite-ξ_t, which are multiplied together: the α and β probabilities, the transition probability a_{ij} and the observation probability $b_j(o_{t+1})$.

$$\text{not-quite-}\xi_t(i,j) = \alpha_t(i)a_{ij}b_j(o_{t+1})\beta_{t+1}(j) \tag{3.15}$$

To obtain ξ_t from not-quite-ξ_t, according to the laws of probability [2], not-quite-ξ_t must be divided by $P(O|\lambda)$, the probability of the observation given the model, which is simply the forward probability of the entire utterance (or, alternatively, the backward probability of the entire utterance):

$$P(O|\lambda) = \sum_{j=1}^{N} \alpha_t(j)\beta_t(j) \tag{3.16}$$

This gives the following final equation for ξ_t:

$$\xi_t(i,j) = \frac{\alpha_t(i)a_{ij}b_j(o_{t+1})\beta_{t+1}(j)}{\sum_{j=1}^{N}\alpha_t(j)\beta_t(j)} \tag{3.17}$$

For the estimation of a_{ij} in equation 3.12, the numerator with the expected number of transitions from state i to state j is then the sum over all t of ξ. The denominator with the total expected number of transitions from state i is the sum over all transitions from state i. Finally, the formula for $\widehat{a}_{ij}$ is:

$$\widehat{a}_{ij} = \frac{\sum_{t=1}^{T-1}\xi_t(i,j)}{\sum_{t=1}^{T-1}\sum_{k=1}^{N}\xi_t(i,k)} \tag{3.18}$$

[2]Since $P(X|Y,Z) = \frac{P(X,Y|Z)}{P(Y|Z)}$

Next comes the formula for computing the observation probability $\widehat{b}_j(v_k)$, the probability of a given symbol v_k from the observation vocabulary V, given a state j.

$$\widehat{b}_j(v_k) = \frac{\text{expected number of times in state } j \text{ and observing symbol } v_k}{\text{expected number of times in state } j} \tag{3.19}$$

This requires the probability $\gamma_t(j)$ of being in state j at time t, which is again computed by including the observation sequence in the probability:

$$\gamma_t(j) = P(q_t = j|O, \lambda) = \frac{P(q_t = j, O|\lambda)}{P(O|\lambda)} \tag{3.20}$$

The numerator is simply the product of the forward probability and the backward probability:

$$\gamma_t(j) = \frac{\alpha_t(j)\beta_t(j)}{P(O|\lambda)} \tag{3.21}$$

To compute $\widehat{b}$, the numerator is the sum of $\gamma_t(j)$ for all time steps t in which the observation o_t is the symbol of interest v_k, and the denominator is the sum of $\gamma_t(j)$ over all time steps t. The result is the percentage of the times to have been in state j and to have seen symbol v_k:

$$\widehat{b}_j(v_k) = \frac{\sum_{t=1 s.t. O_t=v_k}^{T} \gamma_t(j)}{\sum_{t=1}^{T} \gamma_t(j)} \tag{3.22}$$

Where the notation $\sum_{t=1 s.t. O_t=v_k}^{T}$ means the sum over all t for which the observation at time t was v_k.

Using $\widehat{a}$ and $\widehat{b}$, the transition probability A and the observation probability B can be re-estimated from an observation sequence O, assuming that a previous estimate of A and B already exists. These re-estimates are the core of the iterative forward-backward algorithm. The forward-backward algorithm starts with an initial estimate of the HMM parameters $\lambda = (A, B)$ and then iteratively goes through two steps: the expectation step (E-step) and the maximization step (M-step). In the E-step, the expected number of state occupations γ and the expected number of state transitions ξ are calculated from the prior probabilities A and B, respectively. In the M-step, new A and B probabilities are calculated using γ and ξ. Listing 3.3 shows the Python pseudocode for the forward-backward algorithm.

Although in theory the forward-backward algorithm can perform completely unsupervised learning of the parameters A and B, in practice the initial conditions are crucial to the result since they form the starting point for the algorithm. Thus, the HMM structure is often manually initialized or pre-trained, and only the emission probabilities B and (non-zero) transition probabilities A are trained from a set of observation sequences O.

```
def forward_backward_algo(observations of len T,
output vocabulary V, hidden state set Q):
    initialize A and B
    iterate until convergence
        E-step
            $\gamma_t(j) = \frac{\alpha_t(j)\beta_t(j)}{\alpha_T(q_F)}$    $\forall t$ and $i$

            $\xi_t(i,j) = \frac{\alpha_t(i)a_{ij}b_j(o_{t+1})\beta_{t+1}(j)}{\alpha_T(q_F)}$    $\forall t, i$ and $j$
        M-step
            $\widehat{a}_{ij} = \frac{\sum_{t=1}^{T-1}\xi_t(i,j)}{\sum_{t=1}^{T-1}\sum_{k=1}^{N}\xi_t(i,k)}$

            $\widehat{b}_j(v_k) = \frac{\sum_{t=1 s.t. O_t=v_k}^{T}\gamma_t(j)}{\sum_{t=1}^{T}\gamma_t(j)}$
    return A, B
```

Listing 3.3: Python pseudocode for the forward-backward algorithm.

3.3. Dimensions of Music

Music theory is a complex and vast subject. There are various practices, disciplines, and concepts that form musical compositions. Simplified, music can be said to be built by the interplay of melody, rhythm and harmony. Together with texture and dynamics, these form the five most important fundamental elements to create music. Since music modeling is about finding ways to appropriately model these five building blocks, they are introduced in the following subsections (3.3.1 - 3.3.5), which are based on the findings of [West 14] and [Schm 13]. Since this thesis focuses on musical improvisations, a brief summary of what constitutes an improvisation is given in the last subsection 3.3.6.

3.3.1. Melody

Melody is the linear, horizontal representation of pitch. *Pitch* is the sound vibration produced by an instrument or voice and indicates how high or low a note sounds. The sequence of these pitches arranged in a musical phrase creates a melody.
The distance between two musical pitches, measured in number of semitones, is an *interval*. Intervals can be the basis of melody when notes are sounded in succession, this is called melodic intervals. When several notes sound simultaneously, they are called harmonic intervals, better known as chords, which form the basis of harmony (cf. 3.3.3).
The interval between a note and the next higher or lower pitch of the same note is called

an *octave*. There are 12 *semitones* in the octave: A, A♯/B♭, B, C, C♯/D♭, D, D♯/E♭, E, F, F♯/G♭, G, G♯/A♭. These pitches repeat in the same order throughout the range of human hearing, as they do on the keyboard of a piano.
Melodies are usually derived from the traditional major and minor *scales* of tonal music. There are twelve *keys*, derived from the 12 semitones. They indicate which notes in a scale are sharp (♯) or flat (♭), and which note represents the tonal center of a melody.

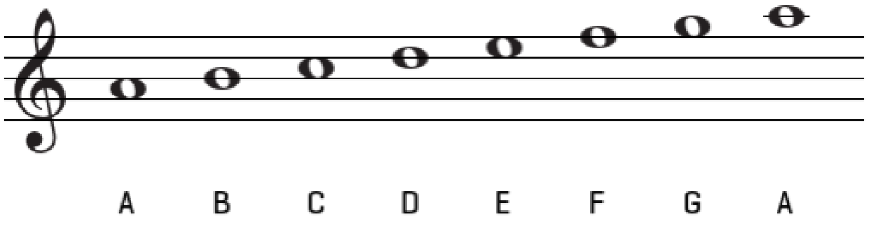
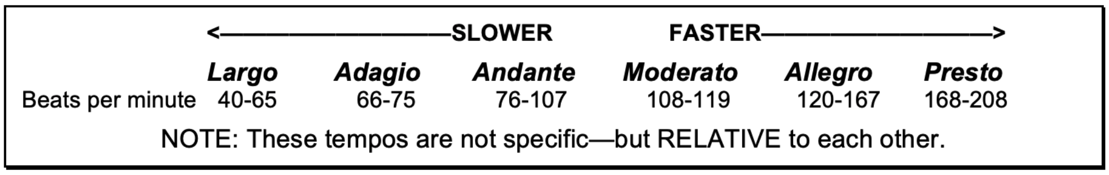


Figure 3.2.: The A minor scale (treble clef) from [Pian 21].

For example, a song in the key of A minor uses notes from the A minor scale shown in Fig. 3.2. Melodies can be described by their type of melodic motion as conjunct or disjunct.
In *conjunct* melodies, notes move in small intervals by whole or half steps. In *disjunct* melodies, there are larger interval jumps between notes.

3.3.2. Rhythm

Rhythm is the element of time in music. The mere succession of pitches does not make a melody. Each note played also has a *duration*. These durations are divided into bar sections, also called note values, such as whole notes, half notes, quarter notes, and so on. The relationship between the durations refers to the rhythm.
There is often an underlying recurring *beat* in rhythm, which is the structural rhythmic *pulse* of the music. But rhythm is not just a constant periodic beat. Rather, the beat serves as a musical skeleton, and rhythm is what results from the combination of notes and *rests* (silences) of varying durations, which sometimes coincide with the beat and sometimes do not.



<———————————SLOWER FASTER———————————>

	Largo	***Adagio***	***Andante***	***Moderato***	***Allegro***	***Presto***
Beats per minute	40-65	66-75	76-107	108-119	120-167	168-208

NOTE: These tempos are not specific—but RELATIVE to each other.

Figure 3.3.: Tempo indications designated by the Italian terms from [West 14]: Largo = “large” or labored (slow), Adagio = slow, Andante = steady “walking” tempo, Moderato = moderate, Allegro = fast (“happy”), Presto = very fast.

The *tempo* measured in beats per minute (BPM) indicates the speed of the beat and is often designated by Italian terms as shown in Fig. 3.3.
Beats can be accented or unaccented, which refers to the intensity or emphasis of notes in that beat. Accented beats can occur at regular or irregular intervals. The organization of beats into recognizable or recurring accent patterns is called *meter*. Figure 3.4 shows the different meters with its time signature and conductor pattern.

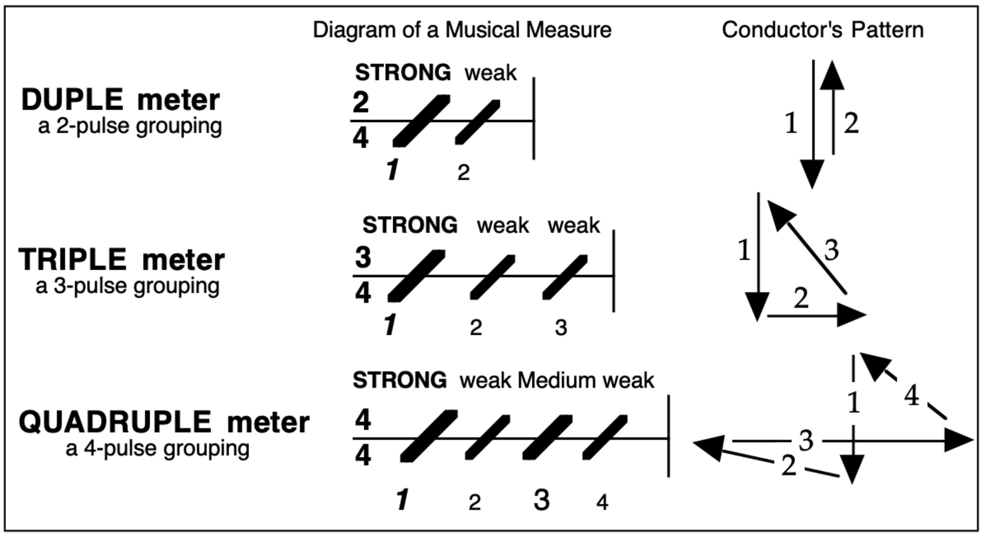


Figure 3.4.: Meter types and its related conductor's pattern from [West 14].

A *time signature* is two numbers, one on top of the other. The numerator describes the number of beats in a bar, while the denominator describes of what note value a beat is (ie, how many quarter notes there are in a beat).

3.3.3. Harmony

Harmony is the verticalization of pitch. It is the relationship between different tones and voices played simultaneously to create a new (appealing) sound. Among other things, harmonies combine two or more pitches into *chords*. These chords are usually arranged in sentence-like patterns called *chord progressions*, and thus support or complement the melody. An example of a chord and a chord progression can be seen in Figure 3.5.

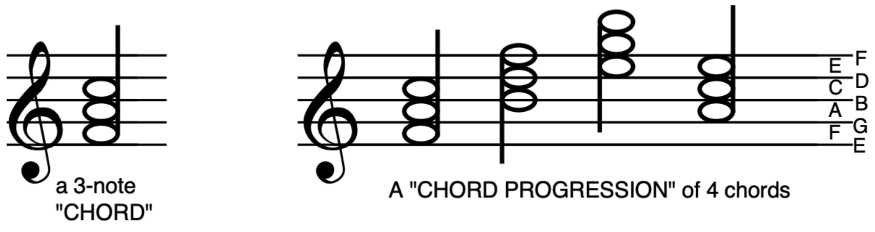


Figure 3.5.: Chords and chord progressions from [West 14].

There are two main types of harmonies, distinguished in terms of their relative harshness: *dissonance* and *consonance*. Dissonant (harsh) harmonies add notes that sound instable when played together. Dissonant chords create musical tension that is often released by resolving to consonant chords. Examples of dissonant intervals are seconds, sevenths, and ninths. Consonant (soft) harmonies sound stable and pleasing. All notes in a consonant chord have intervals that play well together. Examples of consonant intervals include unisons, thirds, fifths, and octaves. Musicians combine consonant and dissonant harmonies to make music more exciting.
It is also harmony when several instruments are played simultaneously. Often there are leading instruments that play melodies (like the voice, wind instruments, etc.) and at the same time others (like the piano) that accompany them.

3.3.4. Texture

Texture refers to the number of individual musical lines, the melodies, and the relationship these lines have to each other. Be careful not to confuse the number of musical lines with the number of performers producing the musical lines. Graphical representations of the different textures can be found in Fig. 3.3.4.

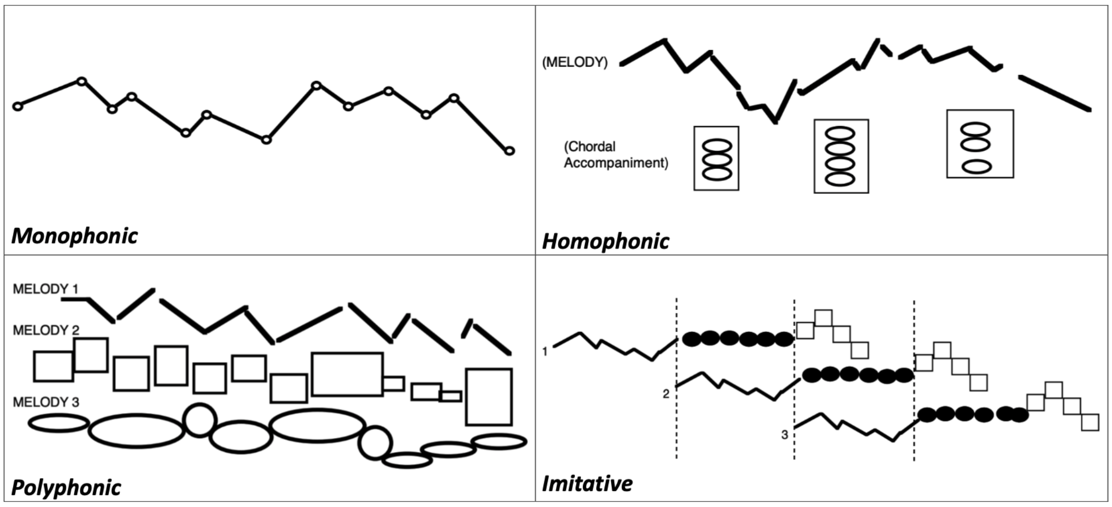


Figure 3.6.: Graphical representation of different textures: monophonic texture (TL), homophonic texture (TR), polyphonic texture (BL) and imitative texture (BR). Combined from [West 14].

The *monophonic* texture is a single-note texture with only one note sounding at a time in music, having no harmony or accompaniment.
When two or more notes are sounding at the same time, but there is generally a prominent melody in the upper part, supported by a less intricate harmonic accompaniment underneath, it is a *homophonic* texture. The underlying accompaniment in homophonic music is often based on homogeneous chords. A special type of homophonic texture is

vocal leading, where voice is accompanied by chords.
Music with two or more independent melodies sounding at the same time is called *polyphonic* texture. The most intricate types of polyphonic texture, canon and fugue, may introduce three to five or even more independent melodies simultaneously.
Imitative texture is a special type of polyphonic texture produced whenever a musical idea is echoed from voice to voice. Although imitation can be used in monophonic styles, it is more prevalent in polyphonic art-music.

3.3.5. Dynamics

Dynamics refers to relative loudness or quietness of music. Sounds can vary in loudness, with dynamics and dynamic potential depending on the shape, size, and material of the sound source, as well as the method of sound production. Changes in dynamics can occur suddenly or gradually. With an *accent*, a note is struck harder to emphasize it temporarily. A dynamic that is getting gradually louder is called a *crescendo*, while a dynamic that is gradually getting quieter is called a *diminuendo* (or decrescendo).
Dynamics play an important role in musical expression, providing an important source of unity and variety, and helping to define musical form. Table 3.3 gives an overview of the most common dynamics, including their symbol, italian term, description, and associated MIDI velocity, where the velocity values are more indicative than exact values.

Table 3.3.: Overview dynamics and velocities from [Vand 12].

Symbol	**Term**	**Definition**	**Velocity**
ppp	pianississimo	extremely soft	20
pp	pianissimo	very soft	31
p	piano	soft	42
mp	mezzo-piano	moderately soft	53
mf	mezzo-forte	moderately loud	64
f	forte	loud	80
ff	fortissimo	very loud	96
fff	fortississimo	extremely loud	112

3.3.6. Musical Improvisation

According to an article in the Encyclopedia Britannica [Brit 12], improvisation, also called extemporization, in music is the improvised composition or free performance of a musical passage, usually conforming to certain stylistic norms but unconstrained by the prescribed features of a particular musical text. It is the creative activity of immediate

("in the moment") musical composition that combines performance with communication of emotion and instrumental technique, as well as spontaneous response to other musicians [Goro 03].
The basis of improvisation is dialogue: with the musicians, with the audience, with the music itself. Jazz works, for example, often take the form of a conversation between instruments. In a sense, it is about the collaborative musical act in which the environment plays an essential role. Soloists take turns contributing to the creation of a cohesive piece, and the many variables involved such as performer mood and ambient acoustics mean that it is impossible to play the exact same solo twice [Hida 17]. In jazz music, improvisation is considered the heart of the style.
Jazz pieces are often lengthy creations that are largely improvised, with the performers knowing only the general structure of the piece, such as the key, the main phrases, or the form. The changes in direction that occur during the course of the piece due to the many variables are impossible to predict. Therefore, the musicians must constantly pay attention, interact, and react.

Improvisation between instruments is not to be confused with accompaniment, whereby the melody or main themes of an instrumental piece are harmonically or rhythmically supported. Since this work focuses on the creation of musical improvisations, the generation of an autonomous melody instead of an accompaniment is aimed at. Furthermore, the melody should pursue an interaction with the improvisation partner by not only copying musical elements of the counterpart, but also by picking them up and generating an individual reaction with them.

3.4. Representation of Music with MIDI

This section introduces ways to represent the musical dimensions from the previous section 3.3. Since this approach focuses on representation using MIDI, it is discussed how to determine the relevant musical information from this format.

The Musical Instrument Digital Interface (MIDI) format is a technical standard for exchanging musical control information between electronic instruments and for playing, storing and editing music digitally [Swif 97]. Since it is also widely used and uses a symbolic representation of music, MIDI lends itself to extracting, representing, and exchanging musical information in this context.

One common melodic representation is the scientific pitch notation (SPN), which specifies musical pitch by combining a note name (A to G), with accidentals if necessary, and a number indicating the octave of pitch (0 to 8 for the standard 88-key piano). Pitch information can also be represented in MIDI numbers from 0 to 127, where the range from 21 to 108 includes all pitches of the standard piano [Vand 12]. The MIDI numbers can be converted to SPN and vice versa, with the assignment as in A.1. However, the pitch pattern of a melody can also be represented using melodic intervals between consecutive notes instead of direct pitch information, as Sanchez Hilgado [Hida 17] shows in his thesis.

One way to represent rhythm is with the representation of note durations and rests. An obvious symbolic representation of the duration is with note values (e.g. a quarter note), which can be obtained from MIDI files as follows. Timing in MIDI files is based on ticks and beats, where a beat corresponds to a quarter note. Beats are divided into ticks, which is the smallest time unit in MIDI (cf. Fig. 3.7).

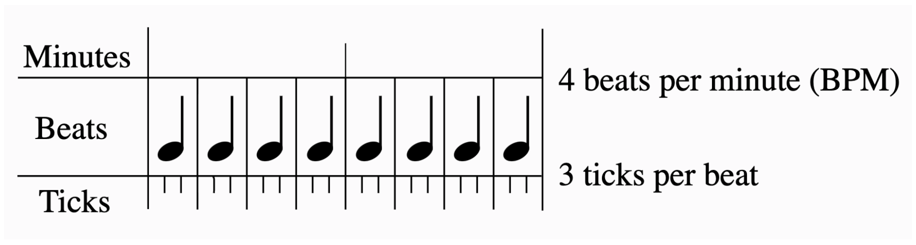


Figure 3.7.: Relation of MIDI ticks, beats and minutes from [Bjor].

Each message in a MIDI file has a delta time, which indicates how many ticks have passed since the last message [Bjor]. The duration of a pitch is obtained by comparing the timing of its NOTE ON and its corresponding NOTE OFF message, where NOTE ON is the event of a key being pressed and NOTE OFF is the event of a key being released [Vand 12]. The length of a tick is defined in ticks per beat, which is stored in MIDI file objects and remains fixed throughout the song [Bjor]. In contrast to the standard tempo indications in music (cf. Fig. 3.3), the tempo in MIDI is not given in beats per minute, but in microseconds per beat. The default tempo is 500000 microseconds per beat, which corresponds to 120 beats per minute, and the default time signature is 4/4. The searched note values are therefore divisors or multiples of the MIDI tempo.
Representations with classical time units are also suitable, whereby the microseconds per tick are the tempo divided by the ticks per beat. For example, the note duration in milliseconds is calculated from the delta time in ticks as in equation 3.23 [Lin 16].

$$\text{note duration in ms} = \frac{\text{tempo} \cdot \text{ticks}}{\text{ticks per beat} \cdot 1000} \tag{3.23}$$

Sanchez Hidalgo [Hida 17] combines the duration with intervals for the duration distance between notes to represent rhythm patterns.
As mentioned in the beginning, besides pitch durations, the usage of rests and its durations are also relevant for rhythm. Rests are obtained from MIDI by calculating the delta

time between the last NOTE OFF and the following NOTE ON message, regardless of the pitches [Maro 97].

In MIDI files, dynamics is represented by the velocity in the NOTE ON and NOTE OFF messages [Vand 12]. It indicates how hard a key is struck, resulting in the effect of a note getting louder or quieter. The velocity value can be any positive integer from 1 to 127 covering the range from a practically inaudible note to the maximum note level. It is essentially the same as the scale of nuances found in musical notation, as in Tab. 3.3. There is a special case when the velocity is set to zero. The NOTE ON message then has the same meaning as a NOTE OFF message, namely turning off the note.

The simultaneity of notes for the harmonic and textural representations is again obtained by the timing of MIDI messages. Notes sound together if their NOTE ON messages follow each other before one of their NOTE OFF messages occurs. If the delta time of the successive NOTE ON messages is 0, the notes are played exactly at the same time [Lin 16], as is the case with chords, for example. Texture representation can be also done by different midi tracks, where each musical voice is represented by one track.

3.5. Related Work

This work extends and combines existing approaches that pursue music modeling and generation using Markov chains and Hidden Markov Models. In this section, the related works are introduced, with the approach of Alvin Lin [Lin 16] and the approach of Thayabaran Kathiresan [Kath 15] being most relevant to the thesis.

3.5.1. Music Generation with Markov Chains

In [Lin 16] Lin processes individual MIDI files into a Markov chain in his program. Therefore, all MIDI NOTE ON messages in the tracks are isolated and organized in a graph of progressions. For each note, its duration is rounded to the nearest 250 milliseconds and it is grouped with any note that was played at the same time. A representation of the resulting directed graph is shown in Figure 3.8.
This graph is represented as an adjacency matrix, storing which nodes of the graph are connected by an edge and how often a transition from a particular note to another note occurs. It has one row and one column for each node, resulting in a $n \times n$ matrix for n nodes. The notes are represented here as MIDI numbers and combined with their durations. For simplicity, only the duration of the note being transitioned to was considered. This approach does not support rest handling. Internally, the Markov chain is represented as a dictionary of dictionaries. Lin uses the Python module *collections* for this, to track the notes and their transitions with *defaultdicts* and *counters*.

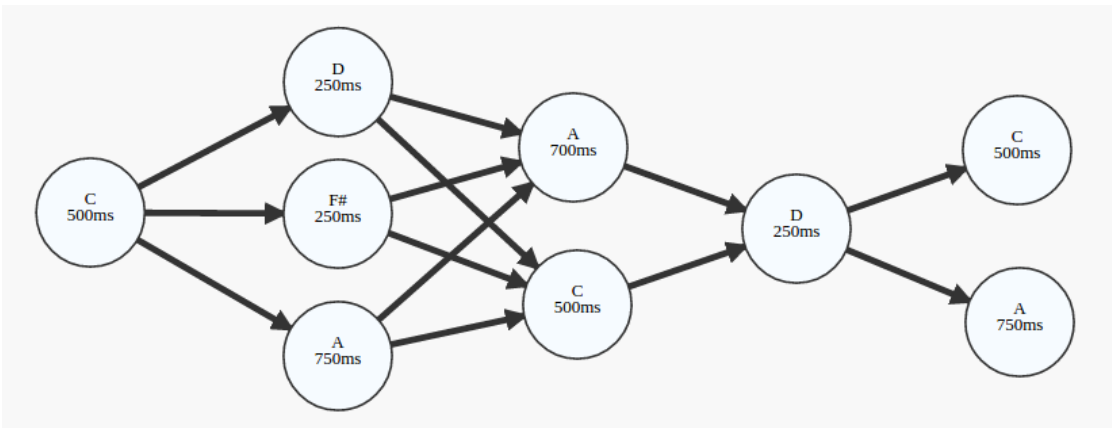


Figure 3.8.: Directed graph of notes and its durations.

To create music, he starts with a random note, looks it up in the matrix, and then selects a random note to transition to, weighting the more frequent transitions.
In this way a monotonous melody is generated, which gives a hint of the original input melody, but also shows that a lot of information is lost due to the simplified representation. The generated melody is saved as a MIDI file, there is no implementation for live input and output. The code of Lin's approach served as the basis for the project and was adapted for Hidden Markov Models.

3.5.2. Music Generation with HMM and Categorical Distribution

In his work investigating automatic melody generation with AI algorithms, Kathiresan [Kath 15] presents a model that handles both notes and rhythm together. This model combines a HMM with Categorical Distribution (CD), so that the HMM is then divided into two main parts. One part is the MM which is responsible for the notes and the other part is the output distribution (in this case it is CD) which is responsible for the rhythm.
CD is a generalized Bernoulli distribution and describes the outcome of a random event that can take one of K possible categories, where the probability of each category is specified separately and the probabilities of all categories sum to 1.
The parameters of CD are thus:

- K - Number of possible outcomes or categories ($K > 0$)
- $p_1, p_2 ... p_K$ where p_i is the probability of category with $\sum p_i = 1$

For the rhythm generation problem, a separate CD is needed for all unique pitches and rests (all rests are treated as one symbol, 'R') present in the reference music. Rests, just like pitches, can have different durations, but Kathiresan does not distinguish in

rest durations and assigns a quarter rest to all of them. This results in a one-category CD for the rest state, where the probability of a quarter note is 1. For each CD, its parameters are extracted from the given training set (rhythm). This approach of using a separate CD for each note or pause limits the algorithm to create a new duration value in the generation process. The CD for a given symbol is calculated by first counting its number of occurrences in the training melody sequence. Then all identical durations for the selected symbol are grouped and counted separately. The probability of a category (duration) for a given symbol is then the normalization of the number of durations with the total number of the selected symbol. This is done for each available symbol in the training melody sequence.
The resulting CD is combined with the trained MM (via melody) by treating it as an output distribution matrix of the MM corresponding to the parameter B from the section 3.2. The parameters Q and A are extracted from the pitches and rests of the training set. Figure 3.9 shows the HMM with CD generative model for a given reference music with its parameters.

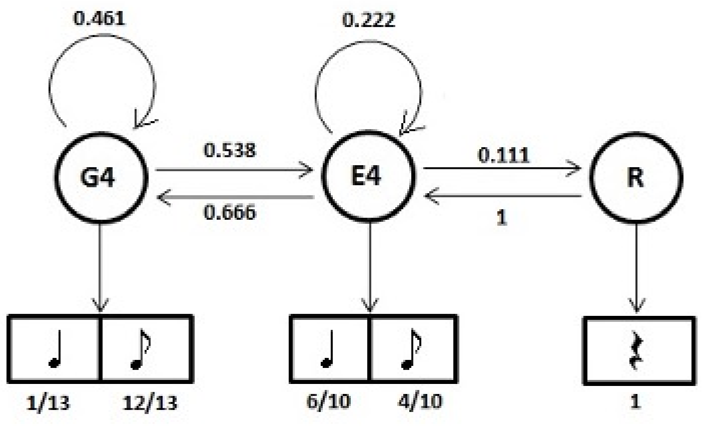


Figure 3.9.: HMM with Categorical Distribution.

The model shown above is able to imitate the given style of the reference melody. The user must specify a length of output sequence (number of notes) that the model should generate. This model is designed to control the number of notes in the output sequence, but not the duration in time. It clearly shows that the presence of a symbol in the generated sequence may or may not be present during further generation. However, there is no chance to get a symbol that was not present in the reference melody.

3.5.3. MySong - Automatic Accompaniment for Vocal Melodies

MySong is a creative tool, introduced in a paper by Microsoft 2008 [Simo 08], that automatically selects chords to accompany a vocal melody, allowing a user with no

musical training to quickly create accompanied (homophonic) music.
At its core, the tool uses a HMM to represent a chord sequence and its relationship to a melody. This HMM essentially represents which melody notes frequently co-occur with each chord type, and which chords typically precede and follow other chords in the database.
The model is trained using a large database of lead sheets [3] of popular music. As pre-processing steps, Simon et al. simplify each extended chord in the database to its core triad and transpose all songs in the database to a single key (C) without loss of generality. In this way, they obtain a set of chords for the parameter Q (cf. 3.2), distinguishing between 62 chord types [4].
First, to learn the statistics that determine the transitions between chords, regardless of the melody, the system examines the chord transitions in each song in the database and counts the number of observed transitions from each chord type to each other chord type. This results in a chord transition matrix, corresponding to A (cf. 3.2), of dimension 62×62, where each cell represents the number of transitions between the corresponding two chord types in the database. The individual rows of this matrix are normalized to obtain the chord transition probabilities.
In the next step the statistics are learned that determine which notes are associated with each chord type. Since the chord sequences in the database are non-overlapping sequences in time, the time period in which a chord is played can be observed and the total duration of each musical note that occurs in the melody fragment during that time period can be counted. These summed note durations are represented in the melody observation matrix, corresponding to B (cf. 3.2), of dimension 60×12 [5], which contains the total duration of each observed note over each chord for all songs in the database. Each row of this matrix is normalized so that each cell represents the observation probability of a pitch looking at a time in which a particular chord is played.
The process of generating chords to accompany a new melody makes the following two assumptions:

1. The vocal track being analyzed was played along with a computer-generated beat and therefore had a consistent tempo; i.e., there is no need to extract timing information from the vocal track.

2. Chords are generated at a fixed interval corresponding to a certain number of beats at the known tempo, which they refer to as “measure”.

For each measure in the recorded melody, there is a 12-element observation vector that counts the occurrences of the 12 musical notes. The likelihood that the observed dis-

[3] A lead sheet contains a melody and the associated chord sequence.
[4] This refers to the 12 root notes times the 5 triads, plus 2 virtual chord types representing the beginning and end of a song.
[5] 60 rows, one for each chord type, except for the “song start” and “song end” chord types and 12 columns, one for each of the 12 musical notes.

tribution of notes occurred under the assumption that a chord is played, is computed by forming the dot product of the observation vector with the logarithm of the corresponding row of the melody observation matrix. This gives the log-likelihood for that chord. Together with pitch tracking, the chord probability calculation is performed once for each melody recorded by the user. Only after that, a chord sequence can be selected in real time for each measure, while the user can also adjust various parameters.

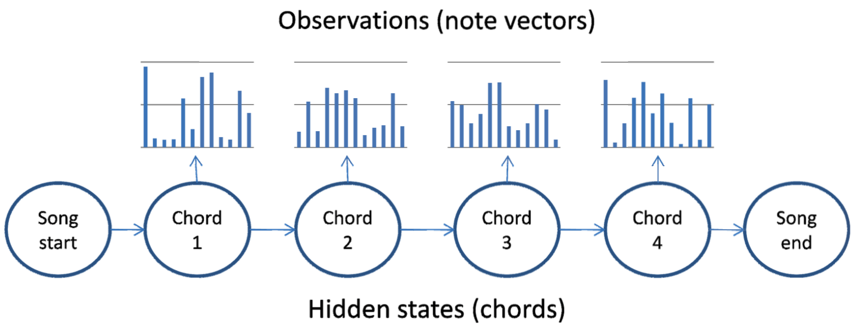


Figure 3.10.: Graphical representation of the HMM used by MySong.

An HMM, as shown in Figure 3.10, is used to choose the best chord sequence, where the hidden states Q correspond to the chords and the observations O correspond to the note vectors at each measure. The state transition probabilities A are drawn directly from the chord transition matrix, while the observation probabilities B are calculated as described previously. This procedure for chord generation must be repeated for all twelve keys, since in practice there is no reason to assume that the melody is in the key of C. The key with the highest overall probability of the chord sequence is selected.

3.5.4. Jazz Improvisation with Markov Chains

In his dissertation, Marom [Maro 97] identifies important aspects of jazz improvisation and presents several techniques based on Markov chains to model these aspects and generate simulations based on human-generated input. Marom models the musical aspects of melody, rhythm, and dynamics, each in separate Markov chains. He refers to the various Markov chains as "simple" in the sense that they are at the top level of the fitting and simulation process. This means that there is no external factor influencing the behavior of these chains and, moreover, there is no interaction between the different sub-models of the final model. They act independently of each other, so that an assignment of melody, rhythm and dynamics is more or less random. Figure 3.11 shows the final model consisting of four Markov chains.
The first one is a rhythmic Markov chain responsible for distinguishing between a note and a rest. This is achieved by extracting a sequence of notes and rests from the music

Notes/Rests:	N	→ N	→ N	→ N	→ N	→ R	→ R	→ N	→ R	→ N
Pitch:	F	→ G	→ G	→ B	→ C		⟶	A	⟶	G
Duration:	$\frac{1}{4}$	→ $\frac{1}{2}$	→ $\frac{1}{8}$	→ $\frac{1}{8}$	→ $\frac{1}{4}$		⟶	$\frac{3}{8}$	⟶	1
Volume:	80	→ 85	→ 80	→ 80	→ 90		⟶	85	⟶	70

Figure 3.11.: Separated Markov chains for melody, rhythm and dynamics.

excerpt without regard to the actual pitch. The Markov chain is then fitted to this sequence of states, whose state space has two states: “N” for note and “R” for rest.
The second chain models the melody meaning the state space consists of the possible pitches. The limitation of this approach, as shown in Figure 3.11, is that there is a transition between notes that are actually separated by rests. This means that the pitch sequence does not “hear” the rests in the music.
For rhythm modeling, in addition to the note-rest differentiation, a Markov chain is used to map the sequence of note durations. The state space here consists of multiples of the smallest note duration.
The volume (dynamics) is represented by the velocities at which the notes are played. They form the fourth Markov chain of the model. As the velocities can be any positive integer, they are rounded to multiples of five to reduce the size of the state space. Just like pitches, durations and velocities are assigned to note events only, skipping the rests. The smallest note duration serves as a timestamp and the various events discussed so far are interpreted at each multiple of the smallest note duration. The smaller the timestamp becomes, i.e. the smaller the note durations used, the more realistic the overall representation of the music in the model becomes.
For each of the submodels, the simulation starts with a random initial state. At each iteration, the entry of the current state in the transition probability matrix is then used to determine the next state. To do this, the sample function takes a sample of entries that have associated probabilities and uses a random number generator along with this distribution of probabilities to select one of the entries.
To allow external events to influence the improvisation, Marom introduces an HMM that he calls the “Controlled Markov Model”, since both hidden states and observations are observable. He therefore speaks of underlying and simple states instead of hidden states and observations. The underlying states represent only external events, e.g. drum patterns that try to “drive” the improvisation, while the focus is still in the simple states, which refer to the Simple Markov Chains approach and represent the musical events. Sampling is then performed for each underlying state that occurs in the original music. The Controlled Markov Model gave the approach a better sense of structure and increased its applicability in the context of an ensemble. It was also intended to introduce a degree of interdependence between aspects of the music that cannot be assumed to

function independently. Although statistically the music was affected by the control signal, this degree was only slight in the results.

3.5.5. Pattern-based Jazz Improvisation with Markov Chains

Sanchez Hidalgo[Hida 17] presents a pattern-based algorithm developed by Norgaard et al. [Norg 13] for the generation of tonal improvisations. This algorithm employs Markov chains to randomly generate both melodic and rhythmic patterns using probabilities extracted from the analysis of the corpus of jazz saxophonist. To find out what the states are, the corpus is analyzed, taking into account both pitch and rhythm. Thus, they obtain pitch patterns of intervals between successive notes in semitones (cf. Fig. 3.12), as well as rhythm patterns of duration and spacing between notes, and create a list of these parameters for all 5 successive notes. This means that they have interval

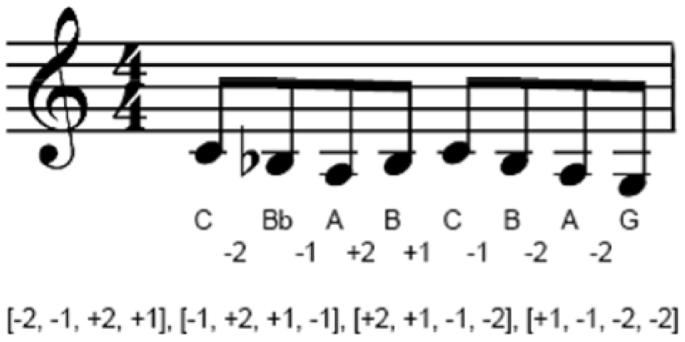


Figure 3.12.: Intervals for melody patterns: A sequence of notes and the distance in semitones between them. Below, the pitch patterns that can be found in the sequence.

vectors for both melodic and rhythmic aspects, which can also be linked together, since each pitch vector is associated with a set of rhythm vectors and vice versa.
Unlike Marom, it allows them to create solos where the pitch and rhythm are not independent. The rhythm and pitch analysis yields two matrices with the respective patterns containing all the information they need to generate a solo. To generate a sequence, the algorithm starts by arbitrarily choosing the initial state and then chooses between one of the adjacent states equally arbitrarily.
For harmony and homophonic texture, Norgaard's original algorithm provided a hard-coded chord accompaniment, which can be modified by Sanchez Hidalgo's extension via a user interface. However, since both approaches work without HMMs, this aspect is irrelevant in the context of this thesis.

3.6. Proposed Approach

The approach proposed in this thesis extracts the key parameters for modeling and generating music from the presented approaches in section 3.5 and combines them into a fully accessible, configurable HMM application for live musical interaction with a player piano.

Music modeling requires finding ways to appropriately map the musical dimensions from section 3.3 to the model parameters from section 3.2. After the essential components of music have been mapped to a model, there are then several ways to generate music using that model. For Markov models, two functions are relevant for the direct generation of music: *predict* and *sample*. While *predict* aims to predict a sequence of states based on observations, this approach uses *sample*, which draws a more or less random sequence of states with corresponding observations from the model. This work aims at live adaptation and generation of music in the context of Spirio Sessions. In live adaptation, continuous learning plays an important role, which is achieved by regularly fitting the model. For live generation, continuous sampling is performed.

3.6.1. Representation of Music Elements

Instead of limiting itself to one possible representation of the musical elements for melody and rhythm, this approach uses a selection of representations for notes and their durations.

One melodic representation is with MIDI numbers, which covers the entire keyboard range of a standard 88-key piano in the midi number range [21, 108]. In this representation the octave information is already integrated.

Another melodic representation, where the octave information has to be calculated separately or retrieved from the input, is the one with semitones. This covers the range of one octave with the 12 semitones from C to B.

The third melodic representation presented borrows from Sanchez Hidalgo's [Hida 17] approach and attempts to capture melodic patterns using intervals between notes. The proposed interval representation covers a range of two octaves with integer values from [-12, 12]. With intervals, the HMM learns the spacing of pitches measured in semitones at each note change.

One rhythmic representation method is the one with classical time units in milliseconds. The note durations are calculated as suggested in Eq. 3.23 and then rounded, extending the rounding approach of Lin [Lin 16] so that the durations are not only rounded up but also rounded down. Instead of mapping the rounded values to the nearest 250 milliseconds as Lin does, the quantization is adjustable in this approach. It is derived from multiples of a quantization parameter, as given in the examples of Table 3.4.

Table 3.4.: Examples of the quantization value and the resulting note duration interval.

Quantization value	**Note duration interval**
50	[50, 100, 150, 200, 250, 300, 350, 400, 450, 500, 550, 600, 650, 700, 750, 800, 850, 900, 950, 1000, 1050, 1100, 1150, 1200, 1250, 1300, 1350, 1400, 1450, 1500, 1550, 1600, 1650, 1700, 1750, 1800, 1850, 1900, 1950, 2000]
125	[125, 250, 375, 500, 625, 750, 875, 1000, 1125, 1250, 1375, 1500, 1625, 1750, 1875, 2000]
250	[250, 500, 750, 1000, 1250, 1500, 1750, 2000]

The table shows that the resulting note duration interval ends at a defined maximum value, which is two seconds by default. It is thus assumed that a note can last a maximum of two seconds, which of course does not correspond to reality. This value is an estimate and was introduced to reduce the state space and still cover most rhythmically relevant values for the given application framework. Since the maximum value max is a transfer parameter, the limit can be set arbitrarily. The note duration is generally mapped to the nearest multiple of the quantization parameter q in a range from q to max.
Since velocity was introduced last, there are no different representations for it, but only the one proposed in [Maro 97]. Here, the range of values from Tab. 3.3 is decomposed into multiples of five to obtain the gradations of velocity and reduce the state space.

3.6.2. Mapping Music Elements to HMM Parameters

In Kathiresan's [Kath 15] HMM with CD, the melodic elements are modeled in the states and the rhythmic elements in the observations. However, the proposed approach of this master thesis does not want to get fixed even in the assignment of the musical elements to the HMM parameters Q and O. Therefore an HMM architecture was developed, to which arbitrary states and observations can be passed, resulting in multiple possibilities of an assignment, which is also called layout in the system. The layout determines which musical events are assigned to the states and observations and in what way. An overview of the different proposed layouts as simplified illustrations can be found in Fig. 3.13.
The assignment of Kathiresan is taken as the default layout and from this a second layout results directly by its inversion, i.e. now the rhythmic symbols are in the states and the melodic symbols in the observations.
In addition, there is the joint layout that offers the possibility to combine several musical elements in one symbol. For example, a model was created that has joint notes and durations in the observations and a set of states without any assignment. The unassigned states of the process could then perhaps correspond to an abstract state of mind, such as mood and feeling.

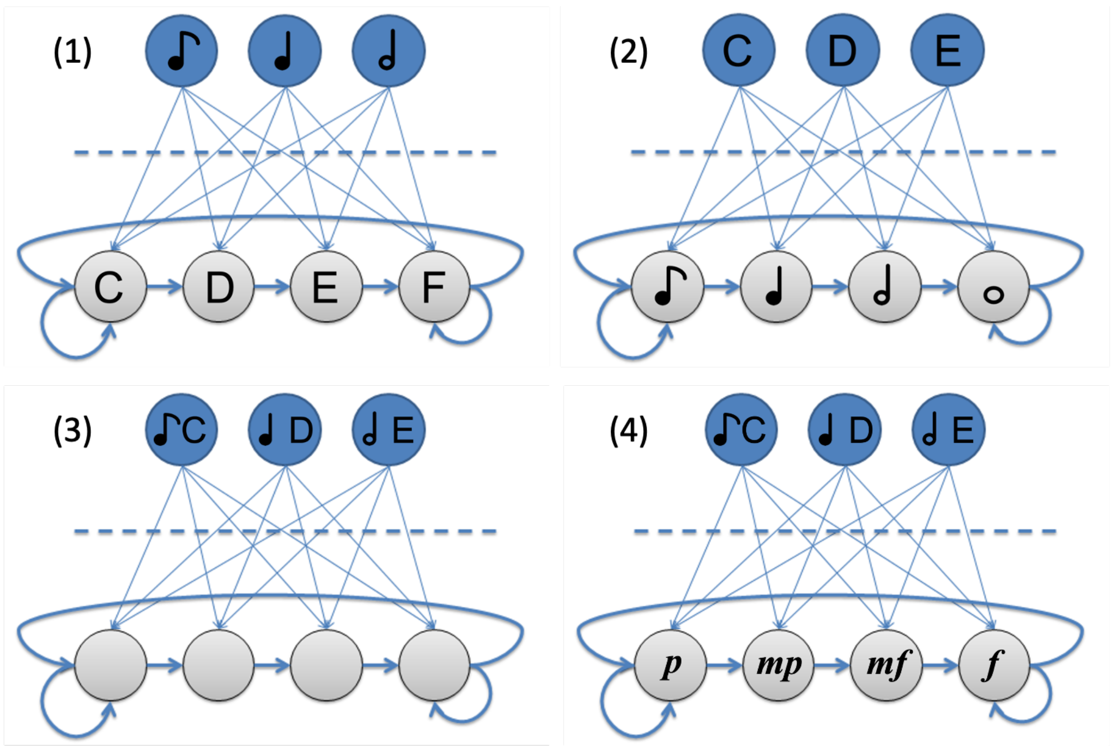


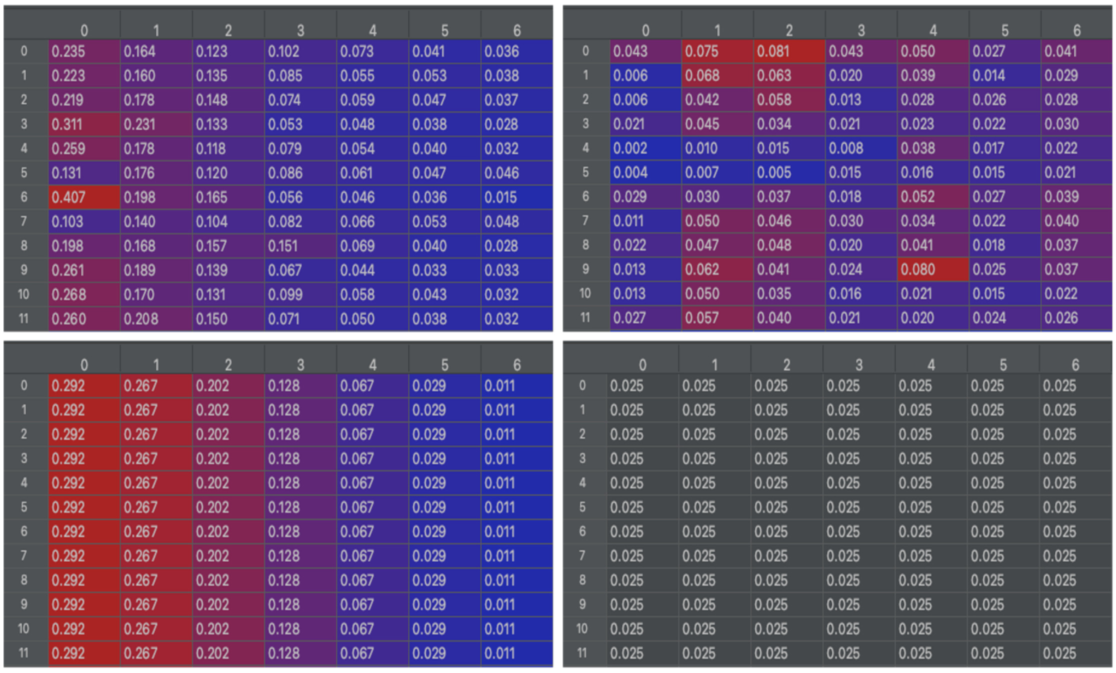
Figure 3.13.: Simplified representation of the proposed HMM layouts with note-duration layout (1), duration-note layout (2), unassigned-joint layout (3) and velocity-joint layout (4). Background HMM from [Souz 10].

To give the states in the joint layout an assignment as well, velocity was used in another layout. The velocity-joint layout keeps the joint observations as before and sets velocity to the states, thus bringing melody, rhythm, and dynamics together in one model.
These are the models that can be used so far. By inversion and further joint combinations of the musical elements, numerous other layouts can nevertheless be created. For example, the hidden states could consist of joint symbols of the output and the observations of joint symbols of the input to model an adequate accompaniment.

3.6.3. Initialization of HMM Parameters

After showing in the first two subsections how the HMM parameters Q (states) and O (observations) were modeled, this subsection deals with the initialization of the remaining parameters A (transition probabilities), B (emission probabilities) and π (initial state probabilities). As mentioned in section 3.2.3, the initial conditions are very important for the learning algorithm and form the starting point for it. This approach proposes different types of initialization, some of which are exemplified in Figure 3.6.3.
Assuming that the number of states N and the number of observations M are given, one obtains an N-dimensional list for the initial state probabilities, an $N \times N$-dimensional

matrix for the transition probabilities, and an $N \times M$-dimensional matrix for the emission probabilities to be initialized.



	0	1	2	3	4	5	6
0	0.235	0.164	0.123	0.102	0.073	0.041	0.036
1	0.223	0.160	0.135	0.085	0.055	0.053	0.038
2	0.219	0.178	0.148	0.074	0.059	0.047	0.037
3	0.311	0.231	0.133	0.053	0.048	0.038	0.028
4	0.259	0.178	0.118	0.079	0.054	0.040	0.032
5	0.131	0.176	0.120	0.086	0.061	0.047	0.046
6	0.407	0.198	0.165	0.056	0.046	0.036	0.015
7	0.103	0.140	0.104	0.082	0.066	0.053	0.048
8	0.198	0.168	0.157	0.151	0.069	0.040	0.028
9	0.261	0.189	0.139	0.067	0.044	0.033	0.033
10	0.268	0.170	0.131	0.099	0.058	0.043	0.032
11	0.260	0.208	0.150	0.071	0.050	0.038	0.032

	0	1	2	3	4	5	6
0	0.043	0.075	0.081	0.043	0.050	0.027	0.041
1	0.006	0.068	0.063	0.020	0.039	0.014	0.029
2	0.006	0.042	0.058	0.013	0.028	0.026	0.028
3	0.021	0.045	0.034	0.021	0.023	0.022	0.030
4	0.002	0.010	0.015	0.008	0.038	0.017	0.022
5	0.004	0.007	0.005	0.015	0.016	0.015	0.021
6	0.029	0.030	0.037	0.018	0.052	0.027	0.039
7	0.011	0.050	0.046	0.030	0.034	0.022	0.040
8	0.022	0.047	0.048	0.020	0.041	0.018	0.037
9	0.013	0.062	0.041	0.024	0.080	0.025	0.037
10	0.013	0.050	0.035	0.016	0.021	0.015	0.022
11	0.027	0.057	0.040	0.021	0.020	0.024	0.026

	0	1	2	3	4	5	6
0	0.292	0.267	0.202	0.128	0.067	0.029	0.011
1	0.292	0.267	0.202	0.128	0.067	0.029	0.011
2	0.292	0.267	0.202	0.128	0.067	0.029	0.011
3	0.292	0.267	0.202	0.128	0.067	0.029	0.011
4	0.292	0.267	0.202	0.128	0.067	0.029	0.011
5	0.292	0.267	0.202	0.128	0.067	0.029	0.011
6	0.292	0.267	0.202	0.128	0.067	0.029	0.011
7	0.292	0.267	0.202	0.128	0.067	0.029	0.011
8	0.292	0.267	0.202	0.128	0.067	0.029	0.011
9	0.292	0.267	0.202	0.128	0.067	0.029	0.011
10	0.292	0.267	0.202	0.128	0.067	0.029	0.011
11	0.292	0.267	0.202	0.128	0.067	0.029	0.011

	0	1	2	3	4	5	6
0	0.025	0.025	0.025	0.025	0.025	0.025	0.025
1	0.025	0.025	0.025	0.025	0.025	0.025	0.025
2	0.025	0.025	0.025	0.025	0.025	0.025	0.025
3	0.025	0.025	0.025	0.025	0.025	0.025	0.025
4	0.025	0.025	0.025	0.025	0.025	0.025	0.025
5	0.025	0.025	0.025	0.025	0.025	0.025	0.025
6	0.025	0.025	0.025	0.025	0.025	0.025	0.025
7	0.025	0.025	0.025	0.025	0.025	0.025	0.025
8	0.025	0.025	0.025	0.025	0.025	0.025	0.025
9	0.025	0.025	0.025	0.025	0.025	0.025	0.025
10	0.025	0.025	0.025	0.025	0.025	0.025	0.025
11	0.025	0.025	0.025	0.025	0.025	0.025	0.025

Figure 3.14.: Examples of different initial distributions of emission probabilities. It shows in each case a section of the emission probability matrix initialized with pre-training and conjugate prior (TL), pre-training and flexible type (TR), Gaussian distribution (BL) and discrete uniform distribution (BR). Horizontally, the halftones are shown and vertically, only the first seven durations are shown due to space constraints. The cells are colored from blue (low probability) to red (high probability). Pre-training was performed on the Jazz MIDI dataset.

The first type of initialization is based on the approach of MySong [Simo 08], where the probabilities are pre-trained from training data. For this purpose, all probabilities are first set to zero and then calculated using maximum likelihood estimation (MLE). The initial state probabilities π are obtained by separately counting the occurrences of all states in the training set and then normalizing the resulting values so that they sum to 1. A similar procedure is followed for the transition and emission probabilities, except that here the state transitions for A and the co-occurrences of the states with observations for B are counted up from the training set. Again, this is followed by normalization, this time regarding the matrices, which are normalized row by row.
Since it is very unlikely that certain notes will appear with certain durations, some combinations of notes and durations will have no observed data. The same applies to state transitions; probably not all of them will be obtainable from the training set. For this reason, a conjugate prior was introduced as in [Simo 08]. This adds "imaginary" instances of each state for each observation, and in this approach it also adds "imagi-

nary" transitions between each state. These imaginary values are very small relative to all observed values in the database. This has the effect of removing zeros in the matrices and smoothing the distribution somewhat. In this case, it also allows to simulate a quasi non-fully connected HMM with the used Python architecture from section 4.4.1, where the transitions actually exist, but are extremely low weighted.

Initialization can also be done with a random, Gaussian or discrete uniform distribution for A, B, and π. With the random distribution, additional trends can be defined to emphasize certain ranges of random values, and also with the Gaussian distribution, certain ranges can be assumed to be more likely than others. For example, it was defined that the middle notes of the piano keyboard are more likely than very low or very high notes, and that short durations are more likely than very long durations of a few seconds. In the discrete uniform distribution it is assumed that all ranges are equally probable. Again, it is important in all distributions that the values are normalized so that the transition and emission probabilities of a state, as well as all initial state probabilities among each other, sum up to 1. Furthermore, pre-training can be performed on top of these initialization types. This results in numerous combinations for the final initialization of A, B and π.

Finally, a flexible initialization type was introduced, which is pre-trained as described at the beginning of the subsection. The difference to the approaches presented so far is that there is no fixed size list and matrices for the probabilities. All of the five HMM parameters (Q, O, A, B, π) are empty at the start of the pre-training and expand with the training data. A function that has not yet been implemented would be to shrink the parameters again, taking into account a defined occupancy value of the states and observations. This allows an even stronger adaptation to the current input.

3.6.4. Continuous Learning

Continuous learning, also called lifelong learning or online machine learning, is a fundamental idea of machine learning in which models continuously learn and evolve based on the input of increasing amounts of data while retaining previously learned knowledge [Pree 20]. In practice, this refers to supporting a model's ability to learn on its own and adapt in production as new data arrives.

In the HMM application, continuous learning is used for live adaptation of the HMM parameters to players' input. This is done by continuously fitting the model at regular time intervals or at certain events, both of which can be specified by the player. From a musical point of view, it makes sense to use musical events or musical time units, instead of classical time units, to trigger retraining. In this sense, time-based triggering can be based on the musical time unit of a beat and event-based triggering can be based on the musical event of a note.

These triggers initiate the retraining process, which involves updating the model parameters A and B (cf. 3.2). In standard HMM training, the transition probabilities and emission probabilities are determined indirectly by the observations using EM algorithm (cf. 3.2.3). This is called unsupervised learning for HMMs, which is useful when the states are hidden and cannot be observed. In this particular modeling case, however, both the observations and the states are given, since both notes and note durations can be observed in the player's input sequence. This allows the model parameters to be trained directly using the given state alignment and maximum likelihood estimation as proposed for pre-training in section 3.6.3. This can be seen as a kind of supervised learning for HMMs. By continuously invoking one of the two training methods, the HMM adapts live to the current input.
How much it adapts can be determined by a weighting parameter. For this purpose, smoothed weighting was applied as proposed in [Schu 95], where the a posteriori estimates are obtained by interpolating the ML (machine learning) parameters with the preset or pre-learned model parameters. With continuous learning, in each retraining, the a posteriori estimated values of the previous time step become the a priori estimated values of the current time step. In simple terms, to obtain the probability matrix M_t for A or B at the current time step t, the matrix M_{t-1} from the previous step (initialization or retraining) is interpolated with the matrix M_{fit} from the current training step using the weighting parameter w as follows:

$$M_t = (1 - w) \cdot M_{t-1} + w \cdot M_{fit} \quad 0 \leq w \leq 1 \tag{3.24}$$

Thus, in terms of the a posteriori expectation, the more lush the training material on the statistic M_{fit} is seeded, the closer M_t nestles to the ML estimates. This means that for w equal to 0 no retraining takes place at all and only the a priori probabilities are adopted. As w increases, the new matrix M_t approximates the current training material more and more, with equal weighting of M_{t-1} and M_{fit} occurring at 0.5. When w equals 1, the statistics of the current training material are weighted 100 percent, completely replacing the previous model parameters. This results in the HMM forgetting everything it has learned before and adapting completely to the input. A music generation based on this would more or less copy the input.
Another important aspect of training is, of course, the training data itself. Regardless of what is being trained, in continuous learning it matters how much past data is used for retraining. Limiting the data considered can be done by a window of certain size that is placed over the data. In continuous learning, this creates a kind of sliding window that moves over the data over time. This gives an additional control parameter to influence the music generation that correlates with the other parameters. With a small window size and a high weighting parameter, for example, the HMM can be controlled to adapt more to the current theme, while with a very large window size and a low weighting

parameter, it can be controlled to learn more of the overall impression of the piece of music.

3.6.5. Sampling Music

Sampling has different meanings in music and statistics. In music, sampling generally means the synthesis of music. In digital recording, sampling is when the sound wave is sampled at a series of points at equal time intervals along the wave to approximate the full wave [Berg 18]. But also the reuse of a sample of a sound recording in another recording, which can be manipulated in various ways, is called sampling in music. In statistics, however, a sample is the selection of a subset of individuals from a statistical population to estimate or represent characteristics of the entire population [Brit 18].
This subsection refers to both musical and statistical sampling, since the proposed approach takes samples from the HMM and uses them to synthesize music. HMM sampling involves drawing a random state sequence and the corresponding feature matrix from the model. In this way pairs of notes and associated note lengths are obtained ready to be played. The number of samples can be specified to generate a piece of music of a particular length. At the beginning, an entire piece of music was created with a larger number of samples and stored in a midi file, as suggested in [Lin 16].
However, for live improvisation things are more difficult. In addition to continuous learning, continuous sampling must be performed by repeatedly creating samples from the HMM. As for training, there are also time-based and event-based triggers for sampling. The rate at which sampling occurs, as well as the number of samples generated at each time point, in addition to the parameters of continuous learning (cf. 3.6.4), affect the final musical result.

Chapter 4.

Software Architecture

In order to create a software architecture where melody generation and output take place separately in decoupled components, a client-server architecture was created. This allows the generation to take place centrally and the output to take place elsewhere, by sending it to the Spirio grand piano, for example. The software architecture from the co-existing Spirio project by Baumann [Baum 21], who bases his work on the open source project A.I. Duet [Mann 16], served as the basis for this.
A.I. Duet was built by Yotam Mann in cooperation with the Magenta and Creative Lab teams at Google. The JavaScript frontend is built with Webpack and displays a user interface with a JavaScript piano and a piano roll. The player can generate input via this web interface, the keyboard or with a midi input device, which is then played back acoustically and visually by A.I. Duet. Based on the input, MIDI segments are created and sent to a Python backend based on a Flask server that processes the input using Magenta's MelodyRNN model. In this way an accompaniment is predicted and generated which is sent back to the frontend and then played and visualized in the user interface.
Baumann [Baum 21] extends AI Duet's software architecture for the generation of multitrack MIDI with the MusicVAE Model. His work provides a good starting point for this project to adapt and extend it to the use case with Hidden Markov Models. The developed software architecture is presented in the following sections.

4.1. Web Sockets

While A.I. Duet and Baumann use Flask's REST API [Fiel 00] to communicate between client and server [1], this work prefers a permanent connection using Web Sockets [Meln 11]. The reason for this is that WebSockets allow bidirectional, event-driven communication between client and server and are therefore better suited for the real-time improvisation scenario in which the HMM application is used. This allows the client and server to communicate independently so that input and output can be generated simultaneously without missing any tones.

Basically, WebSocket is a computer communication protocol that uses a single TCP [Rey 81] connection to provide full-duplex communication channels. Although WebSocket is different from HTTP [Niel 99], it is compatible with the protocol as it is designed to work over HTTP ports 443 and 80 and supports HTTP proxies and intermediaries. To switch from the HTTP protocol to the WebSocket protocol, the WebSocket handshake uses the HTTP Upgrade header. Unlike half-duplex alternatives such as HTTP polling, the WebSocket protocol enables full-duplex client-server interaction with lower overhead, which facilitates real-time data transfer. This works because the server can send content to the client in a standardized way without being requested by the client first, and because messages can be sent back and forth while keeping the connection open.

Table 4.1.: Version compatibility between JavaScript Socket.IO and Flask-SocketIO from [Grin 18b].

JavaScript Socket.IO version	**Flask-SocketIO version**
0.9.x	Not supported
1.x and 2.x	4.x
3.x and 4.x	5.x

Socket.IO [Rauc 21] is a library that simplifies the use of web sockets. For this Project the client-side library uses the JavaScript version 2.3.1, while the server-side library uses the Python version 4.3.2 provided by Flask [Grin 18b]. Table 4.1 shows the compatibility between the Socket.IO versions. Both components have an almost identical API.

The flowchart in Figure 4.1 provides an overview of client-server communication. It shows in simplified form the processes that are triggered by the user on the client side and how these are processed on the server side. The transfer parameters are labeled at the arrows. On both sides the SocketIO event listeners are marked with *"ON"*.

[1]The REST-based approach is unidirectional, meaning at any given time either the client can communicate with the server or the server can communicate with the client, which works for these two approaches because both input and output are processed in chunks.

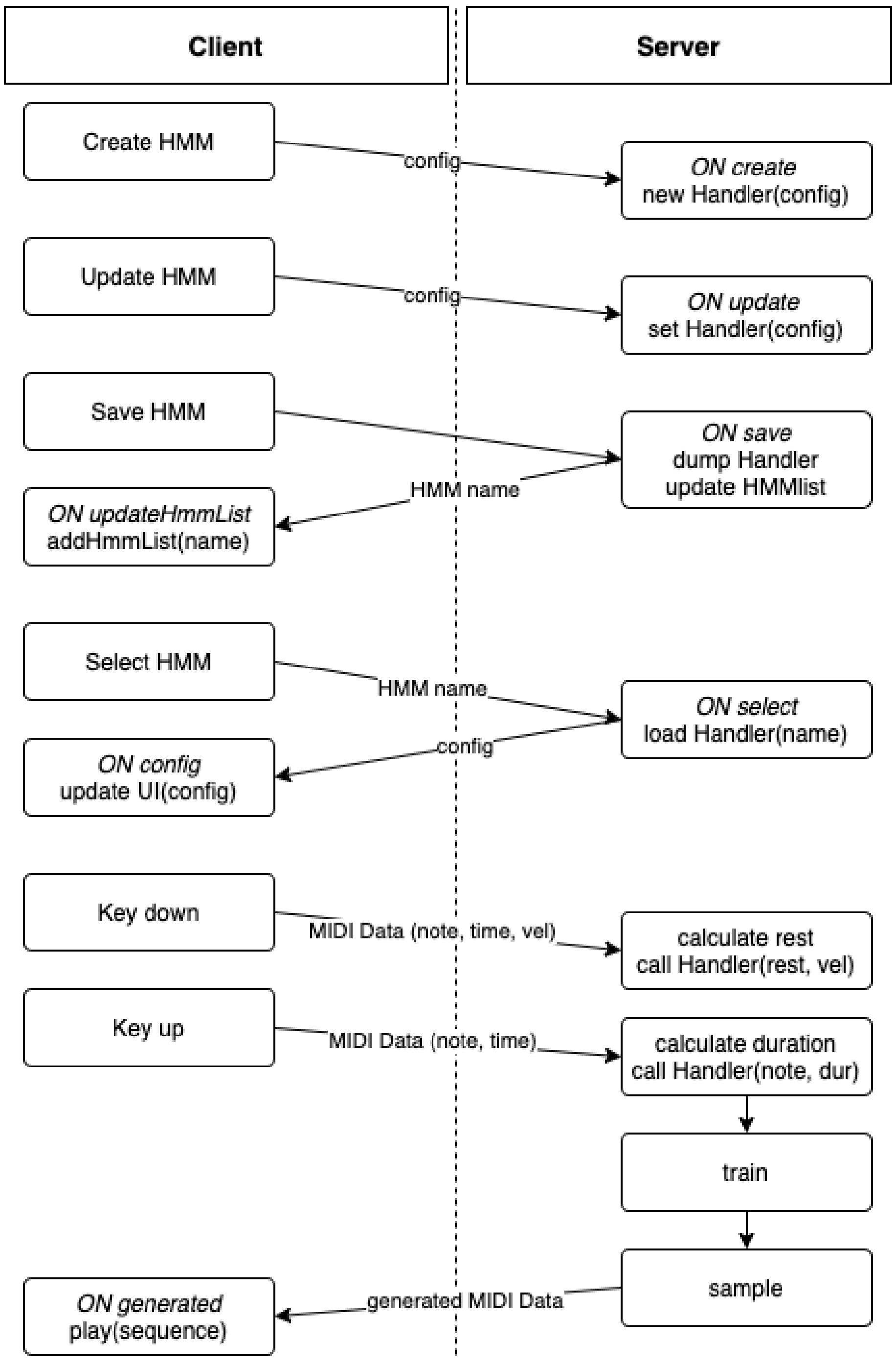


Figure 4.1.: Flowchart for the communication between client and server. Here “Handler” refers to the HMMHandler class from section 4.4.3.

4.2. Backend

The Python backend, based on A.I. Duet, launches a Flask server that communicates with the frontend via SocketIO. Here, in socket events triggered by the frontend, the processes for HMM creation, update and storage are executed, as well as the process for melody processing and generation. To allow multiple users to access the server at the same time, a client-side session handling has been implemented. In a background task a UDP socket is executed that can receive OSC messages. When used in Spirio Sessions, this UDP socket can receive messages with beat information from the connected tool Max.

4.2.1. Flask

The server and the websocket communication in the backend were provided by the Python framework Flask [Grin 18a]. Flask is a micro web framework, which means that it does not require any special tools or libraries. There is no database abstraction layer, form validation, or other components where existing third-party libraries provide common functionality. Since all backend classes were implemented in Python, it also offers the possibility to stay in the same programming language. This minimalistic structure and few dependency usage offers the optimal environment for the development of the application.
However, Flask supports extensions that can add application functionality as if they were implemented in Flask itself. There are extensions for object-relational mappers, form validation, upload handling, various open authentication technologies, and several common framework-related tools.
Flask-SocketIO [Grin 18b] enables Flask applications to access bi-directional, low-latency communication between clients and the server. The client-side application can use any of the SocketIO client libraries in JavaScript, Python, C++, Java, and Swift, or any other compatible client, to establish a persistent connection to the server.

4.2.2. Session Handling

For multiple user access, the Flask extension *request* has been integrated, which adds support for client-side sessions to the application. All clients are assigned a specific room when they connect to the server, named with a unique session ID of the connection. This session ID can be retrieved with *request.sid*. A client can join arbitrary rooms, which can be given arbitrary names. When a client disconnects from the server, it is removed from any rooms it has joined. The *socketio.send()* and *socketio.emit()* context-free functions also accept a *to*-argument to send to all clients in a given room. For addressing a message

to a single client, its personal session ID can be used in the *to*-argument.
As soon as a client is connected to the server, the client's session ID is queried from the server. This is used as key in a global cache dictionary entry, while the associated value contains another dictionary. The value dictionary manages the key-up and key-down events, as well as the HMM handler and the HMM list for a session. As soon as the client disconnects, the cache entry with the corresponding *request.sid* is removed from the dictionary.

4.2.3. Model Storage

Once a user has configured or trained an HMM, he may want to save it to continue working with it at a later time. The storage of the HMM on the server was realized with *pickle* [Pyth 21]. The *pickle* module implements binary protocols for serializing and de-serializing a Python object structure. Serialization ("pickling") is the process of converting a Python object hierarchy into a byte stream, and de-serialization ("unpickling") is the reverse process of converting a byte stream, from a binary file or bytes-like object, back into an object hierarchy.
In a created pickle directory on the server subdirectories are added for each client with it's session ID, where the serialized HMM handler classes are stored. The advantage of storing the whole HMM handler class and not just the HMM itself is that the class also stores the interface configurations, as well as the list of all played tones. Thus it can be said that the complete current state of the user is stored at the time of saving.
With the command *pickle.dump* the class is serialized, passing the object and the file in which it should be saved. For this purpose, a file is created in the pickle directory beforehand, which is then passed to the method. The name of this file consists of the string "HMM" plus a timestamp that is generated at the moment of saving. The "HMM"-string is just a placeholder that is passed by the client and can be easily replaced in the future with a string from a user input text field for the file name. The timestamp then still guarantees that the file name remains unique. De-serialization is done with *pickle.load*, where only the relative path to the binary file is passed.
Once the client is connected to the server, a global HMM list is created by reading the file names from the pickle directory. This list is sent to the client to make the stored HMMs available for selection on the interface. It was originally implemented that the directory of a client is deleted from the server as soon as the client disconnects. This was because the session ID is unique and expires after disconnection. Therefore, the directory can no longer be accessed afterwards. For a permanent storage the session IDs were therefore ignored for the time being and all HMMs are stored in the parent directory. In the future a user administration could use usernames instead of session IDs to manage the subdirectories.

4.2.4. Beat-based Triggering from MaxMSP

In the Spirio Sessions, musicians such as saxophonists and pianists improvise with the Spirio grand piano. For the intelligent melody generation of the grand piano, the various projects are connected to it via computer. There can be additional parameterization from the outside using the Max MSP[2] tool, such as setting the MIDI velocity.
Thus, the idea came up that the HMM program could receive the beats for beat-based triggering directly from Max. The advantage of this is that, compared to beat estimates in the tool, estimation inaccuracies are avoided and the beat generation can be controlled more specifically. Beats can be generated in Max using a metronome or a beat detector, which can be switched on and off as desired.
With each beat, Max sends Open Sound Control (OSC) [Wrig 03] messages to a specific port using the User Datagram Protocol (UDP) [Post 80], which provides a datagram mode for packet-switched computer communication. On the HMM server a UDP socket listens to this port in a background thread, which is started as soon as beat-based triggering is enabled in the frontend. For each received message containing a beat the melody processing is executed. This way, additional parameters, e.g. for retraining, could also be received from MaxMSP later.

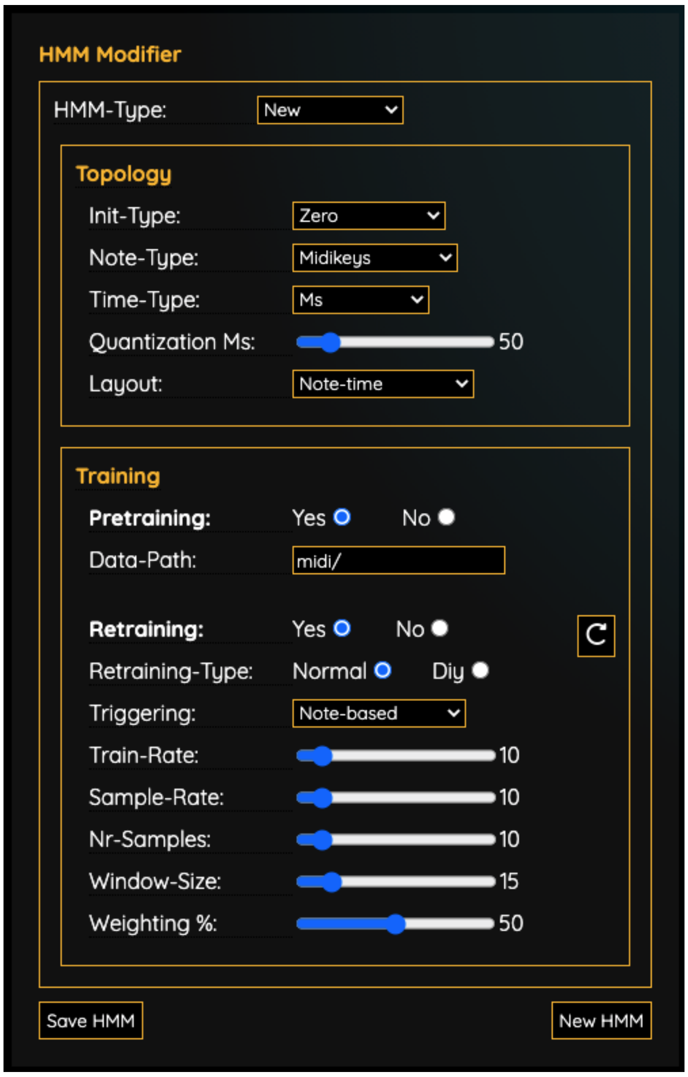
4.3. Frontend

Baumann's [Baum 21] extension of A.I. Duet served as the basis for the front end, providing a good starting point for building the target system. This allows selecting an MIDI input and output port for improvisation with the Spirio grand piano to receive and send MIDI via the Web MIDI API of the browser.
Baumann also added, using the JavaScript package *MidiConvert*, the recording of played and generated MIDI information, each of which can be stored in different tracks of a MIDI file. This can be used to make MIDI recordings for better tracking and documentation in Spirio Sessions, for example.
In the context of this thesis, the frontend from Baumann's project was extended by an user interface that integrates various configuration parameters. These serve to modify both the core HMM, as well as the music generation process, of the program and thus to influence the resulting improvisations. This way, different HMMs can be created, stored and updated live during the Spirio Sessions. The HMMs configuration user interface is embedded in Baumann's frontend and can be shown and hidden by a toggle button. Further details of the individual areas are described in the following.

[2]Max, also known as Max MSP, is a visual programming language for music developed and maintained by the software company Cycling '74 [Cycl 20].

4.3.1. Configuration User Interface

Figure 4.2 shows a screenshot of the configuration user interface, which is called the “HMM Modifier”. Here the user can interact with the system by adjusting the parameters that affect the music modeling and music generation from section 3.6.



HMM Modifier

HMM-Type: New

Topology

Init-Type: Zero
Note-Type: Midikeys
Time-Type: Ms
Quantization Ms: 50
Layout: Note-time

Training

Pretraining: [✓ Yes] No
Data-Path: midi/

Retraining: [✓ Yes] No
Retraining-Type: [✓ Normal] Diy
Triggering: Note-based
Train-Rate: 10
Sample-Rate: 10
Nr-Samples: 10
Window-Size: 15
Weighting %: 50

Figure 4.2.: Screenshot of the configuration user interface.

The design was chosen so that the configuration field fits well into the rest of the user interface and does not interfere with other system interaction. For example, the background is chosen to be transparent and the box can be shown and hidden at any time via toggle button, so that the displayed notes are always visible.
To give the layman information about what the setting of each parameter does, tool tips with short descriptions have been added to them, which appear when hovering over them. Furthermore, UI restrictions have been introduced so that no settings can be made that are not technically possible. For example, if pre-training is disabled and the

initialization type is zero or flexible, the sampling rate must be greater than or equal to the training rate. This is because the HMM is initially empty (flexible) or all probabilities are set to zero for these initialization types. Therefore a training must take place first before anything can be sampled from the HMM. In these cases, warnings are issued by the browser to the user in the form of dialogues, which are displayed until the user dismisses them. These alerts are also used for further notifications, such as confirmation that an HMM has been created, saved or updated.

To be able to display the numerous setting elements clearly also on smaller screens or minimized windows, resizing has been implemented for the UI. So the configuration box automatically shrinks or grows relative to the window size and a scroll bar appears if necessary.

All configuration parameters can be grouped into two sections, Topology and Training, which are visually represented by separate boxes. The parameters assigned to the topology section concern the HMM structure, i.e., the music modeling, from sections 3.6.1 to 3.6.3. These concern initialization, note representation, duration representation, quantization of durations, and layout (assignment of symbols to states and observations) of the HMM.

All parameters assigned to the training section affect pre- and retraining as well as the sampling of the HMM, i.e. the music generation, from section 3.6.4 and 3.6.5. These include the activation and deactivation of pre- and retraining, the pre-training data, the method of retraining (supervised or unsupervised), the basis of triggering, the sampling and training rate, the number of samples generated, the window size, and the weighting parameter. A detailed listing and description of all configuration parameters can be found in the Appendix A.2.

To create a new HMM the HMM type is set to "New". After that the configuration parameters can be set as desired and with the "Create HMM" button a corresponding HMM can be created. In this case, the button behaves like a submit button by which the configuration input values of a form element are read and sent to the backend in JSON format. See Listing 4.1 for an example of a JSON object that contains the selected configuration parameters.

To save a created HMM and the settings of the parameters a "Save HMM" button is available, which triggers the model saving process of section 4.2.3 in the backend. The name of the saved HMM is then returned from the backend and displayed in the frontend's HMM type. A saved HMM can no longer be changed in its topology. However, a new HMM can be created on its basic settings.

During musical interaction with the system, it may be that the parameters for the music generation should be adjusted, but without creating a completely new HMM and thus losing the training progress. For this purpose there is an update button which reads out the retraining parameters and sends them to the backend for updating.

```
{
 hmm-type: "new",
  init: "zero",
  note: "midikeys",
  time: "ms",
  quantisation: 50,
  layout: "note-time",
  pretrain: "yes",
  files: "midi/",
  retrain: "yes",
  train-type: "normal",
  triggering: "note-based",
  train-rate: 10,
  sample-rate: 10,
  nr-samples: 10,
  window-size: 15,
  weighting: 50
}
```

Listing 4.1: JSON object of configuration parameters.

4.3.2. Class Structure

The class diagram in Figure 4.3 visualizes the individual components and the structure of the frontend. From this it can be seen that the frontend's core is the Main class, setting up the web interface and the event-based communication between the other classes and thus providing the entry point.
The websocket-based communication with the backend is handled by the AI class serving as a kind of interface between fontend and backend. The AI class is responsible for handling the API calls, processing the played input and sending it to the backend, as well as receiving, processing and passing the generated output from the backend.
To provide a JavaScript piano interface the Keyboard class uses AudioKeys, which offers a QWERTY keyboard for web applications. This is responsible for handling the key events, that are also triggering the sounds. Additionally the keyboard serves as interface to the MIDI input, triggered by corresponding events of the Midi class.
Mp3 files for all tones are handled by a sampler in the Sound class which acoustically replays the notes of an instrument. As Baumann [Baum 21] implemented multitrack generation the instruments piano, synths, bass and drums are supported, for each of which a sampler is created. However, the HMM application is used for single track

improvisations and therefore only makes use of the piano instrument.
The visual representation of the played notes is instantiated by the Element class, which also provides the piano.
The piano roll is commonly used today to graphically represent music. It visualizes notes and their durations as rectangles on a time grid, which have the note's position and duration's length. The roll class implements such a piano roll for displaying both, the player's input and the AI's output separated by color on screen.
The UI class sets up all the elements for the configurable user interface and exchanges information with the backend via the AI class to create, update, and save HMMs.

4.3.3. Event-driven Architecture

The basis for the event-based communication between classes in the frontend builds the asynchronous and event-driven architecture of the Node.js *event* package [npm 21]. All connections between the particular classes are setup and handled in the Main class. For communication each class has to inherit the EventEmitter class, which enables emitting of events by using the *eventEmitter.emit()* method. This method allows to pass arbitrary arguments to the listener functions. The exposed *eventEmitter.on()* method is then used to attach one or more functions to named events emitted by the respective object. When triggering an event all of its attached functions are called synchronously, in the order they were registered. As shown in Figure 4.3 the classes inheriting from the EventEmitter class are AI, UI, Keyboard, Element and Midi class.

4.3.4. Web MIDI

A special requirement assigned by the Spirio Session project was to be able to send generated output to the Steinway & Sons grand piano as well. This way musicians can interact and play directly with the grand piano in an improvisation session. Because of the widespread use of MIDI in music, most digital devices and musical instruments support the MIDI format and protocol. That's why the standard is used in Spirio Sessions to enable the exchange of musical information between improvisers' instruments, computer programs and the Spirio grand piano.
The Web Audio Working Group of the World Wide Web Consortium (W3C) developed the Web MIDI API [Wils 15] to enable the use of the standard in browsers or browser-based operating systems. Nowadays, most modern browsers support and implement the Web MIDI API, which supports the MIDI protocol and enables web applications to list and select MIDI input and output devices on the client system, and to send and receive MIDI messages. By providing low-level access to the MIDI devices available on users' systems, the API enables both non-musical and musical MIDI applications.

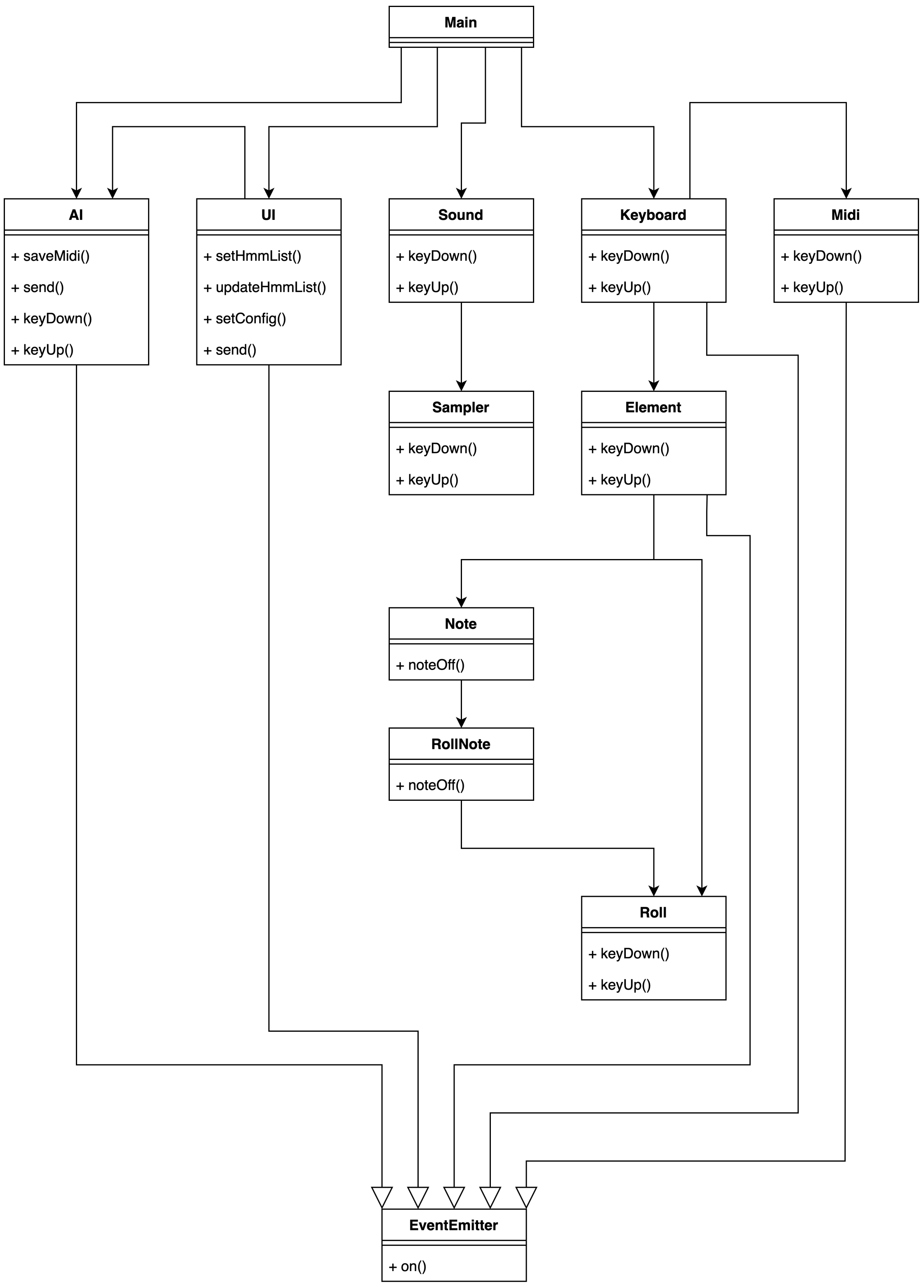


Figure 4.3.: Class diagram of the frontend.

The Web MIDI API is designed to expose the mechanics of MIDI input and output interfaces and the practical aspects of sending and receiving MIDI messages without semantically describing these actions. This requires, for example, performing binary arithmetic when sending and receiving MIDI messages in order to encode or decode MIDI byte streams.
Therefore, the WebMidi.js library [Cote 21] was used for the implementation, which simplifies the use of Web MIDI API for web developers by semantically rewriting its functionality. The library offers functions like playing a specific note and event listeners for incoming notes.
The user interface offers the possibility to select the MIDI input and output port via dropdown menus. A listener is created on the selected MIDI input port and all associated channels, which forwards incoming notes to the web keyboard and thereby the sampler. To the selected MIDI output, the generated notes are sent on the correct channel depending on the instrument. In the case of the HMM implementation only channel 1 is relevant on which the melody for the piano is sent.

4.4. Model Implementation

4.4.1. Core HMM

Hmmlearn [Lee 21] is a Python library for unsupervised learning and inference of Hidden Markov Models. It is open source on GitHub[3] and commercially usable by BSD license. The library provides simple algorithms and models for learning HMMs in Python, following the scikit-learn API as closely as possible, but adapted to sequence data. The architecture is built on scikit-learn, NumPy, SciPy, and matplotlib. There are three different Hidden Markov Models available in *hmmlearn*, as listed in Tab. 4.2.

Table 4.2.: Available Models in hmmlearn

hmm.GaussianHMM	Hidden Markov Model with Gaussian emissions.
hmm.GMMHMM	Hidden Markov Model with Gaussian mixture emissions.
hmm.MultinomialHMM	Hidden Markov Model with multinomial (discrete) emissions.

The HMM application uses the last one in the table, multinomial HMM, to build a model with discrete emission probabilities. The implemented myHMM class inherits from the MultinomialHMM class and extends it with parameters and functions. Listing 4.2 shows the MultinomialHMM constructor with passing parameters. All parameters are set to default values in the implementation except *n_components*, *n_iter* and *init_params*. *n_components* refers to the number of states that is flexibly passed in the constructor

[3]`https://github.com/hmmlearn/hmmlearn`

of the child class (myHMM). The number of EM algorithm iterations is upper bounded by the *n_iter* parameter. The training proceeds until *n_iter* steps were performed or the change in score is lower than the specified threshold *tol*. *n_iter* has been set to 1000 to ensure that the EM algorithm achieves convergence in the given number of steps. In order to not have random initialization of the HMM parameters A, B and π before training, the parameter *init_params* is passed as an empty string. This is because *hmmlearn* performs this initialization by default for all HMM parameters, but the application needs to use the HMM parameters initialized and trained by the child class.

```
class hmmlearn.hmm.MultinomialHMM(
    n_components=1, startprob_prior=1.0, transmat_prior=1.0,
    algorithm='viterbi', random_state=None, n_iter=10, tol=0.01,
    verbose=False, params='ste', init_params='ste')
```

Listing 4.2: Constructor of hmmlearn's multinomialHMM class.

The functionality for training and sampling is already provided by *hmmlearn* as *fit* and *sample* method. A feature matrix of individual samples is passed to the *fit* method to estimate the model parameters using the EM algorithm. To generate random samples from the model, the *sample* method takes the number of samples to generate and returns a feature matrix, as well as the sequence of states generated by the model, of that size. A method of the parent class named *_check_and_set_n_features* had to be overwritten. This method is executed before training in the *fit* method and checks whether the passed feature matrix is in the correct form, i.e. whether it is a sample of the multinomial distribution consisting of non-negative integers. It then adjusts the number of features (*n_features*) of the HMM to the maximum symbol from the feature matrix. However, in continuous learning, the training process must be repeated over and over again with different feature matrices without changing the features themselves. Therefore, this aspect has been adapted so that the *n_features* is not based on the passed feature matrix, but on the *n_features* of the child class.
As an extension of *hmmlearn*'s MultinomialHMM, the myHMM class implements a model that receives the parameters Q and V, as well as an initialization type (cf. 3.6.3), in the constructor. The parameters A, B and π are then set up in the constructor, according to the initialization type passed.
Furthermore, myHMM implements a new training type, which was introduced in section 3.6.4 as supervised learning. In a general training method, it can be chosen whether to apply this supervised learning or the unsupervised learning of hmmlearn to adjust the model parameters. This method also implements the weighting of the a priori and a posteriori probabilities from Eq. 3.24 with a passed weighting parameter.

4.4.2. Parsing Data

The parser is responsible for setting up the states and the observation vocabulary and passing them, along with the specified initialization type, to the myHMM class to create an HMM. When creating the states and the observation vocabulary, the concepts presented in the sections 3.6.1 and 3.6.2 are distinguished. That is, the specified note type, time type and layout are considered to define the state and observation space.
As soon as the HMM has been initialized, pre-training and retraining of the model parameters can be started. Therefore, the incoming MIDI data must be put into the correct format to map it to the states and observations of the HMM. This process must work for MIDI files as well as for live MIDI input data.
To extract important information like pitch, timing and velocity from the MIDI data, the Python library *pretty_midi* [Raff 14] was used. *pretty_midi* contains utility functions and classes for handling MIDI data so that it is in a format from which information can easily be modified and extracted. It is designed to make the most common operations applied to MIDI data as straightforward and simple as possible. *pretty_midi* represents a MIDI file in the hierarchical manner shown in Figure 4.4.

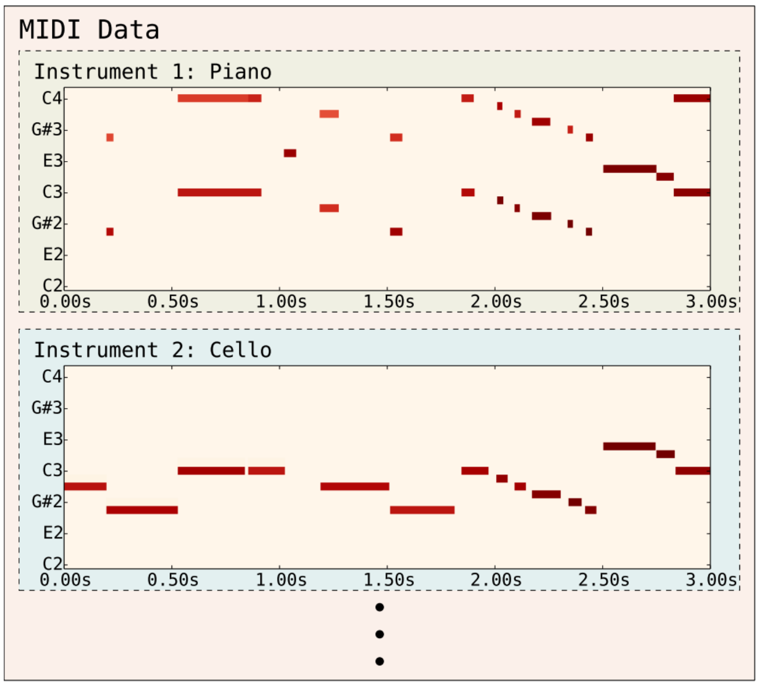


Figure 4.4.: Hierarchical Representation of a MIDI file from pretty_midi [Raff 14]. Here the MIDI data is represented as a list of instruments, each of which has its own piano roll of events.

The module includes functionality for parsing and writing MIDI files, creating and manipulating MIDI data, synthesis, and information extraction. The source code is also supposed to be straightforward to understand and modify and is available on GitHub[4]. *pretty_midi* was preferred over *mido* [Bjor] because it implements the functionality to extract the timing information even from complex polyphonic MIDI files. This is easier than extracting them from the individual midi messages, as would be the case with *mido*. The data parsed by pretty_midi is put into the form of state-observation pairs, again taking into account note type, time type and layout. Such a state-observation pair is created for each note and rest in the input sequence or file. This finally allows the mapping of the data to the model parameters to train the HMM.

4.4.3. Processing and Generating Music

Figure 4.1 shows that on the server side, an HMMHandler class is used to manage all client-side functions related to the HMM. For each connected client a new HMMHandler class is created, which is stored and managed in the cache object of the server. After all, the HMMHandler class is responsible for processing the input and generating the output.

```
def __init__(self, train=True, sample_rate=10, nr_samples=10,
                window_size=15, quantisation=50, layout="note-time",
                train_diy=False, train_rate=10, files="midi/",
                init_type="zero", pretrain=True, weighting=50,
                note_type="midikeys", time_type="ms",
                triggering="note-based"):
```

Listing 4.3: Constructor of HMMHandler class.

Listing 4.3 shows the constructor of the HMMHandler class. Here it receives all configurable parameters of the user interface. These are defined as self variables of the class, which makes them accessible in the server at any time, e.g. to update the parameters. In addition, the parser and thus the core HMM are initialized in the constructor.
The flowchart 4.1 also shows that for each key-up and key-down event a *call* method of the HMMHandler is called, passing different parameters depending on the event.
In the key-up event, the notes and note durations are determined and passed to the handler. Therefore, the corresponding key-down event is looked up in the cache to calculate note durations. It could be assumed that the last key-down event always corresponds to the current key-up event. But this is only the case for monophonic input. For polyphonic input another key-down event may have occurred at the same time or in the meantime. To process both monophonic and polyphonic input, the corresponding pitch in the past key-down events is picked instead. Subsequently, the note duration can be calculated from the timing information of the key-up event and the timing information

[4]`https://github.com/craffel/pretty-midi`

of the corresponding key-down event.
In the key-down event, the rests and rest durations, as well as the velocity, are determined and passed to the handler. Rests actually always refer to the last key-up event, regardless of whether the melody is monophonic or polyphonic. Therefore the timing of the last key-up event in the cache can be used to calculate the rest duration. To avoid that every microsecond results in a rest, only rests in a defined range are considered.
Both notes and rests are parsed together with their durations into state-observation pairs (cf. 4.4.2) and put into a list of the HMMHandler. The list thus contains all past played notes and rests, from which a feature vector with the defined window size can be created for training.
Training and sampling are performed considering the triggering type, as well as the train and sample rate. Either notes or beats are counted which are compared to the rate. If they match, the corresponding process is triggered.
In the sampling process the drawn samples have to be parsed back to MIDI format. The generated MIDI sequence is returned as JSON object, passed to the client and finally played there.

Chapter 5.

Experimental Validation

To validate the software, a subjective, experimental evaluation by the musicians of the project is chosen. In the Spirio Sessions, the musicians play with the developed system and can express comments, impressions and ideas.
The sessions take place at regular intervals to enable continuous evaluation and improvement of the tool. The musicians are project members with a lot of musical experience, as they study or even teach music. There are two trained jazz saxophonists and one trained pianist who have recorded and evaluated material in four Spirio sessions that have taken place so far.
49 experiments were conducted with the HMM in which the musicians matched their expectations with their observations. A detailed table of experiments performed can be found in A.3. This is the current status at the time of writing, but Spirio Sessions will continue beyond this in the future.

5.1. Technical Setup

Figure 5.1 graphically represents the technical setup in Spirio Sessions. It shows that the audio signal of the input instrument is recorded via microphone and forwarded to a beat detector and a pitch tracker. The beat detector examines the audio signal for beats, which it then sends via OSC to the project's first prototype with Markov chain sampling. The pitch tracker analyzes the audio signal and converts the pitches to MIDI format. The central element between the components is a Max/MSP patch that contains, among other elements, the pitch tracker and the Markov chain.
In this work, the right part, the HMM application, was developed, which can be used for music generation besides Markov chain sampling. For this purpose, the MIDI data from the pitch tracker is received in the client of the application and forwarded to the backend. There, the data is processed and the newly generated MIDI data is returned to the client, which in turn passes it on to the Max Patch. In the Max Patch, the generated data gets to a velocity control component to adjust the volume if necessary. Finally, the data is passed from there to Spirio, where it is played back. The Max Patch also passes

the OSC messages from the Beat Detector to the HMM Sever for beat-based triggering (cf. 4.2.4).

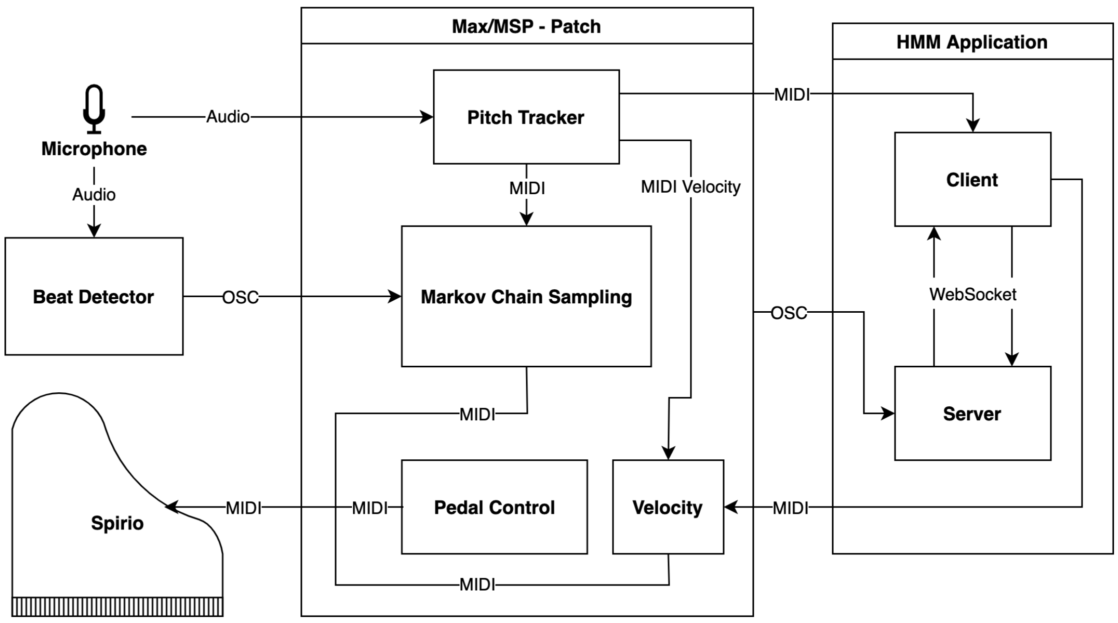


Figure 5.1.: Technical setup of the Spirio Session.

5.2. Results

At the beginning, the sessions were conducted with the first prototype as a command line program, only later the web application was used, which was further developed until the end. Thus, when looking at this table, it should be taken into account that the program was in the process of development and some comments refer to outdated versions. Some of the musicians' evaluations could be technically interpreted and used for improvement of the system. The experiments are referenced in the following with Exp. date/number.

A noticeable aspect at the beginning was the overflow, as mentioned in Exp. 2021-03-25/1. The grand piano continued to play for about the same length of time after the input play was finished. However, the problem of the overflow was solved by adjusting the parameters sample rate and number of samples accordingly.

Another point that was mentioned several times were sound repetitions, which was traced back to the typical HMM phenomenon of self-loops. The architecture of HMMs allows a state to transition to itself, which favors the occurrence of self-loops, where the same state is used multiple times consecutively. This leads to a note being played over and over again. If the self-transitions get a higher probability, the chance for self-loops

also increases. This can be prevented by limiting the number of allowed transitions of a state to itself.
However, an improviser in Exp. 2021-03-25/3 also remarked, after repeated listening, that the note repetitions actually had an interesting sound effect and were very concise, since pianists usually hardly play like that.
In the same experiment, it is worth noting that even at this early stage of the project, it was observed that the rhythm was well captured.

Exp. 2021-03-25/4 was performed with a very low quantization value and Exp. 2021-03-25/5, for comparison, with a very high quantization value. It was noted that a low quantization of 50 ms gave a hectic result with many fast phrases that had not previously occurred at higher values. A too high quantization value of 250 ms resulted in a mechanical sounding melody that was more predictable and did not have rests.
The results from these two experiments are consistent with the expectation of what the quantization value should do (cf. 3.6.1). Low values result in shorter note duration intervals in the HMM, making the melody seem more hectic. Higher values result in longer note duration intervals, mapping most observations to the same duration value and making the melody seem mechanical.
From the comment that the HMM did not have rests, an improvement step could be initiated; rest handling was implemented.

In general, the adaptability of the system to player input plays an important role and is therefore frequently listed in the experiments. In Exp. 2021-04-15/2 and Exp. 2021-04-15/3, it was examined whether Spirio would adapt to a chromatic or diatonic playing style. In these experiments the joint layout was used.
First it was played with blues scale and chromatic, which Spirio partially adopted and also played a relatively similar answer phrase (like variation), but still remained more diatonic. This was maybe due to the pre-trained parameters and showed that in this case the system needed time to adapt to the new input, but it still adapted. At the end Spirio played a really nice blues line, according to the musician.
To see how quickly chromatic passages would disappear in the output, it was then switched to diatonic playing. First Spirio still played some phrases with chromaticism, but then it also became relatively diatonic like the input.

To speed up the adaptation process, either a non-pretrained HMM or a higher weighting of retraining could be used as in Exp. 2021-04-15/4. Here the weighting was increased to 80 percent to see how that would affect the adaptation. Initially, the pianist played only notes from C major, but then made an abrupt change to Gb major. It was observed that the transfer of tonal material was then much faster than in the previous recording, just as expected.

A similar experiment was conducted to see how the note type “intervals” would affect Spirio’s response. For the fastest possible adaptation, a strong weighting was again set, this time of 90 percent. In Exp. 2021-04-15/8 only fourth structures were played, which Spirio adopted more and more, i.e. Spirio actually adopted interval structures after some time.
To check if and how fast Spirio would follow the tonal material, in Exp. 2021-04-15/9 the pure fourths playing was stopped and changed to free tonal material. As expected, Spirio also quickly forgot the fourth structures and followed the new material.
That the weighting parameter also worked in the opposite direction was shown in Exp. 2021-04-15/14. With a weighting of 10 percent, Spirio did not really go into what the saxophonist played.

The fact that the setting of the sampling parameters can have a major influence on the overall result was already indicated at the beginning of this section in the overflow problem. This was also evident in a later series of experiments (Exp. 2021-04-15/15 - 2021-04-15/18).
On the one hand, with the settings in Exp. 2021-04-15/16 everything was very confused, albeit from both sides, or in Exp. 2021-04-15/18 Spirio played many atonal lines.
On the other hand, in Exp. 2021-04-15/15 Spirio adopted the diatonic and was much more active, so the interplay between saxophone and grand piano meshed better and the two voices could develop more. In Exp. 2021-04-15/17, it was even said that Spirio’s scale sounded great and that there was a very hip key change.
This shows that a lot can be done to improve improvisation by adjusting the sampling parameters.

Chapter 6.

Outlook

Even though some concepts for musical improvisation with HMMs have already been developed within the scope of this work, the ideas for them are far from exhausted. In a continuation of this work, further Markov models could be developed, especially for the generation of polyphonic music, which is not yet possible with the current models.
In this context, chords could be modeled as note vectors in the HMM, in addition to single notes. These note vectors could be represented either separately from the monophonic model in a pure polyphonic chord model, or together with the single notes in a kind of heterophonic model. A selection of popular jazz chords is shown in Figure A.2. Mapping the input could be done with a list of defined chords. An alternative would be to define chords by a certain range of distances between simultaneously played notes. This way, it could be distinguished whether it is a chord, if the note distances are within the range, or a melody with accompaniment, if they are outside. This approach could be error-prone as soon as the melody and accompaniment get too close to each other on the keyboard.
Therefore, a left-right hand HMM is also suggested to separate melody and accompaniment. On the piano, the left hand usually plays the accompaniment and the right hand plays the melody. For mapping, a clear boundary could be set on the keyboard, e.g. in the middle, to distinguish what was played by the left hand and what by the right hand. In this way, two separate HMMs could be trained, one for each hand, from which the output could be sampled simultaneously.
Of course, the mapping approaches mentioned so far only make sense for the input of polyphonic instruments like the piano. To create polyphonic output with monophonic input, an approach as in [Simo 08] is more appropriate. So the notes and chords played by the piano could be contained in the hidden states and the notes of the saxophone in the observations. At this point, joint symbols would be needed for both sides, states and observations, to model the notes and durations (and velocities) for both instruments. Recordings of improvisations between saxophone and piano would then be suitable for the pre-training data. Otherwise, a way would have to be found to appropriately map the monophonic input to a polyphonic model.
A separate HMM for velocity would also be conceivable in addition to the HMM for

melody and rhythm. This way, instead of re-sampling it for each note, the velocity could be adjusted in larger intervals. Smooth transitions like crescendo and diminuendo could also be modeled more easily this way. Another idea would be to use an inverted velocity for the output, i.e. if the input plays quietly, the output gets louder and vice versa. This function could be switched on and off with a checkbox in the user interface.
As already mentioned in section 3.6.2, the implementation offers an easy realization of further layout combinations. The user could be free to choose which musical elements (note, time, velocity) to join and how to assign them to the states and observations.
The implementation of the UDP socket for receiving OSC messages, is originally used for performing beat-based triggering (cf. 4.2.4). In the future, this architecture offers the possibility to receive also other parameters from the Max Patch. For example, dynamically generated parameters for retraining could be obtained from the Max Patch in addition to the manual setting in the web interface.
As the Spirio Sessions project is ongoing beyond the scope of this master's thesis and hopefully will continue for a long time, there is no doubt that the above points offer potential for further sub-projects to continue the HMM improvisations project.

Chapter 7.

Summary

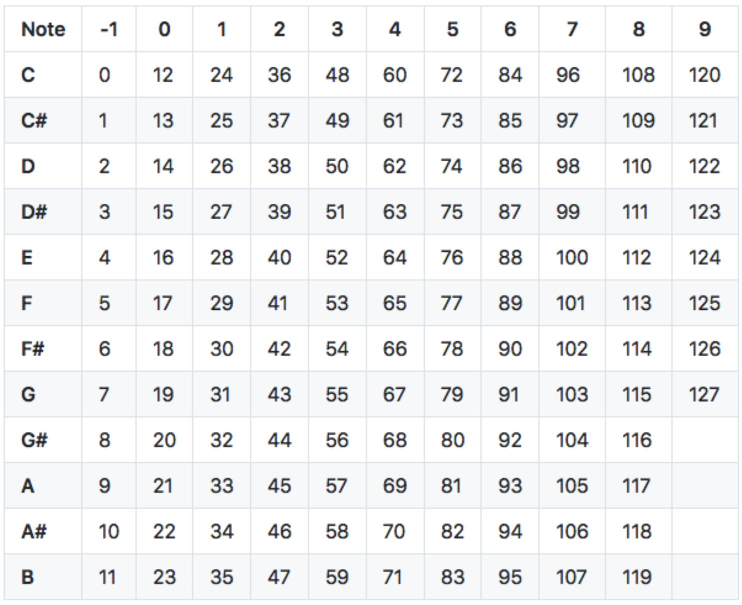
This master thesis took place as part of the Spirio Sessions [Trum 21] project, which aims to explore concepts of free improvisation between humans and machines in different research directions through the development of prototypes, different combinations of software modules and artistic evaluation. To determine whether co-creative processes between AI and human are possible at eye level, the developed approaches to human-machine interaction are explored and evaluated in the so-called Spirio Sessions, where a human musician interacts with the AI-controlled grand piano "Spirio-R".
The focus of this subproject was to investigate automatically generated musical improvisation using Hidden Markov Models (HMMs). Within this framework, an application was developed for an almost fully accessible and configurable HMM that enables experiments in musical improvisation between an AI (the HMM), embodied in the player piano, and a human musician.
HMMs are stochastic models that have the advantage over the deep learning approach in that they require fewer training processes and instead of imitating musical styles, they develop special degrees of freedom. As extension of the Markov chain, the Hidden Markov Model is formally specified by a set of hidden states Q, a sequence of observations O drawn from a vocabulary V, the initial state probabilities π, the transition probabilities A and the emission probabilities B. The decoding problem, the evaluation problem and the learning problem are the three fundamental problems formulated for HMMs. To solve these problems there are three main algorithms, which are the Viterbi Algorithm, the Forward Algorithm and the Forward-Backward Algorithm.
The data for HMM training was self recorded by musicians of the project and the model already achieved convincing results with little training data. Without pre-training, the HMM even adapts more closely to the player input, since its structure is merely built on it. Both pre-training data and live training data are in MIDI format, which has the advantage that the musical information is in symbolic form and is thus easier to process and less error-prone than e.g. the processing of pitch information. Finally, in order to train the HMM on a larger dataset, the Jazz ML ready MIDI dataset [Sai 18] was chosen, which consists of over 935 jazz music tracks in MIDI format.
Simplified, music can be said to be built by the interplay of melody, rhythm and harmony.

Together with texture and dynamics, these form the five most important fundamental elements to create music. Music modeling is about finding ways to appropriately model these five building blocks. Therefore, ways of representing the musical dimensions with the chosen MIDI format and extracting them from MIDI messages were presented.
This work extends and combines existing approaches that pursue music modeling and generation using Markov chains and Hidden Markov Models. The approach of Lin [Lin 16] implementing Markov chains for music generation in Python and the approach of Kathiresan [Kath 15] presenting a musical HMM with CD were the most relevant related works for this thesis.
The proposed approach extracts the key parameters for modeling and generating music from the related works and combines them into a HMM application offering access to modify the core HMM and music generation process. Instead of being limited to a fixed HMM structure, the approach offers a selection of different parameters that influence music modeling. By choosing different representations, layouts and initializations of the model parameters, numerous possibilities arise for the structure of the final HMM.
Once the HMM has been modeled, music can be generated from it, aiming at live adaptation and generation of music in the context of Spirio Sessions. In live adaptation, continuous learning plays an important role, which is achieved by regularly fitting the model. For live generation, continuous sampling is performed. For both training and sampling, additional parameters were introduced that influence the final musical result.
In order to create a software architecture where melody generation and output take place separately in decoupled components, a client-server architecture was created. This allows the generation to take place centrally and the output to take place elsewhere, by sending it to the Spirio grand piano, for example. The architecture provides, among other things, websocket-based communication, session handling, model storage, UDP receiving for OSC messages, web MIDI input and output, and a configuration user interface for changing the HMM parameters.
In the subjective, experimental evaluation of the system in the Spirio Sessions, it was found that the system mostly adapts very well to the player input. It was also shown that atypical but interesting sound effects can be produced by the HMM and that the system is even praised by professional musicians for its “hip tone change”, “good sounding scale” or “nice blues line”.

Appendix A.

Supplemental Information

A.1. Assignment Table SPN to MIDI



Note	-1	0	1	2	3	4	5	6	7	8	9
C	0	12	24	36	48	60	72	84	96	108	120
C#	1	13	25	37	49	61	73	85	97	109	121
D	2	14	26	38	50	62	74	86	98	110	122
D#	3	15	27	39	51	63	75	87	99	111	123
E	4	16	28	40	52	64	76	88	100	112	124
F	5	17	29	41	53	65	77	89	101	113	125
F#	6	18	30	42	54	66	78	90	102	114	126
G	7	19	31	43	55	67	79	91	103	115	127
G#	8	20	32	44	56	68	80	92	104	116	
A	9	21	33	45	57	69	81	93	105	117	
A#	10	22	34	46	58	70	82	94	106	118	
B	11	23	35	47	59	71	83	95	107	119	

Figure A.1.: The assignment chart from [Bene 16] shows the MIDI numbers from 0 to 127 which correspond with all the notes from C0 to G9. C4 is middle C assigned to midi number 60.

A.2. Configuration Parameters

HMM-Type The HMM-Type stands for the core HMM that is used. Either a new HMM can be created or an already saved HMM can be selected.

Topology The Topology section includes all parameters that can be configured to modify the HMM structure, which are the following:

- **Init-Type:** The initialization type for the transition and emission probability matrices can be
  - **zero**, where all cells of the matrices are initialized to zero.
  - **flexible**, where the matrices are initialized empty and grow by training.
  - **discrete**, where all cells of the matrices are initializes with the same probability (normalization).
  - **gauss**, where the matrices are initialized with a Gaussian distribution with an increasing probability for shorter durations and midtones.
  - **random**, where the matrices are initialized with a normalized random distribution.
- **Note-Type:** The assignment to a note type determines how the pitches are represented in the HMM. This can be
  - **Midikeys**, which cover the entire keyboard range of a standard piano with 88 keys in the midi number range from 21 to 108 and integrate octave information.
  - **Semitones**, which cover an octave with 12 semitones from C to B, so the octave change depends on the input.
  - **Intervals**, covering the range of two octaves with 25 intervals from -12 to 12. With intervals the HMM learns the key distances at each note change.
- **Time-Type:** The assignment to a time type determines how the durations of tones and pauses are represented in the HMM.
  - In **ms**, measured from 0 to 2000 ms, where the resolution depends on the quantization parameter introduced in the next point.
  - In **beats**, the standard note length representation in a range from a 32nd note to a whole note at tempo 120.
- **Quantization:** The quantization setting determines the resolution for the note durations in milliseconds. A smaller quantization results in more detailed representations that can distinguish between more note lengths, but also in larger models that need to represent all of these note lengths.
- **Layout:** The layout determines how the states and observations are assigned to the selected notes and time types.

  - For **Note-Time**, the notes are assigned to the states and the durations to the observations.
  - For **Time-Note**, it is the other way around, the durations are assigned to the states and the notes to the observations.
  - For **Joint**, the observations are assigned the notes and durations jointly, while the states are not given a specific assignment. What is learned in the states is unknown and could be something like a musical harmony.
  - For **Velocity-Joint**, as in the Joint-HMM, the observations are assigned the notes and durations jointly, but this time the states also receive an assignment, and that is the velocity.

Training The Training section contains all the parameters that can be configured to control HMM training, which includes pre-training and re-training.

- **Pretraining** The pre-training radio button controls the activation and deactivation of the pre-training process, which is to adjust the probabilities of the HMM from prerecorded midi files.
- **Data-Path** The Data-Path text field expects the directory path to the midi data for pre-training. This can be
  - midi/, the relative path to the prerecorded jazz saxophone midi data.
  - piano/, the relative path to the prerecorded jazz piano midi data.
  - a local directory, referenced by an absolute path, e.g. /Users/mustermann/-Documents/Midi/.
- **Retraining:** The retraining radio button controls the activation and deactivation of the retraining process, which is to adjust the probabilities pf the HMM from live recorded input midi data.
- **Retraining-Type:** There are two types of retraining that can be distinguished:
  - **Normal**, which corresponds to standard training for HMMs using EM algorithm, where state probabilities are learned indirectly from observations.
  - **Direct**, which means direct training of the states and observations by counting and normalizing their occurrences.
- **Triggering:** The triggering type determines the event that triggers the retraining and sampling. The respective event is then taken into account for the train and sample rates, which are explained in the following paragraphs. Triggering can be

  - **note-based**, where each note played represents an event.
  - **beat-based**, where each detected beat represents an event. The beats correspond to the beats received by the OSC messages.

- **Train-Rate:** The training rate determines the period in which the training is triggered, i.e. the parameters of the HMM are updated. Depending on the triggering type, the integer value refers either to the number of tones played or the number of beats detected. For example, the default value of 10 means that training occurs every 10 tones or beats.
- **Sample-Rate:** The sampling rate works like the training rate only that it determines the period in which the sampling is triggered, i.e. tones are generated from the HMM.
- **Nr-Samples:** The number of samples generated means the number of note-duration pairs that are drawn from the HMM and sent to the output as a melody. A higher number often results in greater variety in the overall impression of the melody over time. It should be taken into account that the number of samples must be adjusted to the sample rate, otherwise underflow or overflow may occur. This happens when too few or too many tones are generated for the number of tones played.
- **Window-Size:** The window size specifies how many past input tones are considered for the retraining. For example, with a small window size the HMM can be controlled to adapt more to the current topic while with a very large window size it can be controlled to consider the entire piece of music.
- **Weighting:** The focus on the current theme can also be set by the weighting parameter. This determines the percentage weighting of the retrained matrix in relation to the existing matrix.
  - **0%** means that only the existing matrix is considered, which results in no retraining at all.
  - **50%** means that the existing matrix and the retrained matrix are considered in equal parts, which is the default setting.
  - **100%** means that only the retrained matrix is considered, replacing the existing matrix.

A.3. Experimental Validation Table

Table A.1.: Experimental validation table, Part I.

Date/Time	**Nr**	**Instrument**	**Technical remarks**	**Improv approach**	**Impression**	**Comments**
2021-03-08	1	Alto saxophone	HMM			
2021-03-08	2	=	=			
2021-03-08	3	=	=			
2021-03-08	4	=	=			
2021-03-08	5	=	=			
2021-03-25	1	MIDI-Keyboard	HMM y 15 15 20 190 y y 25	Free, always leaving rests or room for Spirio	Overflow: runs on behind for about the same length	Experiment “overflow”, how long output continues (currently exact function of parameters not yet known)
2021-03-25	2	MIDI-Keyboard	HMM y 15 15 20 190 y y 25	Free	Constant note repetitions are very noticeable; adapts tonal material well, without “replayin” too much	Note repetitions?
2021-03-25	3	MIDI-Keyboard		Free tonal material rests for Spirio	After repeated listening: Note repetitions actually have an interesting sound effect and are very concise, as pianists usually hardly play like this	Rhythm is well captured
2021-03-25	4	MIDI-Keyboard	HMM y 15 15 20 50 y y 25	Free tonal material without a certain specification	Hectic result, many fast phrases but also interesting, as before such fast phrases did not occur	Try low quantization, what effects does this have
2021-03-25	5	MIDI-Keyboard	HMM y 15 15 20 250 y y 25 = Eighth at 120 bpm	Free tonal material without a certain specification	Seems mechanical, more predictable, no rests	Try high quantization

Table A.2.: Experimental validation table, Part II.

Date/Time	**Nr**	**Instrument**	**Technical remarks**	**Improv approach**	**Impression**	**Comments**
2021-03-25	6	Baritone saxophone	HMM y 15 15 20 125 y y 25			
2021-03-25	7	Baritone saxophone, more distance to Spirio	HMM y 15 15 20 125 y y 25			
2021-03-25	8	Baritone saxophone with pedal 80				
2021-03-25	9	Baritone saxophone with pedal 80				Call and response Pedal automated
2021-04-15	1	Functional test				
2021-04-15	2	MIDI-Keyboard	HMM Quantization 150 Layout Joint	Bluescale, Chromatic	Spirio “waits” and responds more to input; plays relatively similar response phrase (like variation)	Layout “Joint” tried for the first time, Pretraining unknown Notes Valentina: F. plays chromatic, S. partly too, but more diatonic; at the end Spirio plays a really nice bluesline
2021-04-15	3	MIDI-Keyboard	= (without reset, same parameters)	Mainly diatonic, no chromaticism if possible	Still plays some phrases with chromaticism, then also relatively diatonic (like input)	Try how fast chromatic passages disappear from output (in reference to Joint Layout) Note V.: Spirio plays very chromatic, F. plays longer phrases, S. still leaves a lot of space
2021-04-15	4	MIDI-Keyboard	<>(Reset) Weighting 80%	First only notes from C major, abrupt change to Gb major	The transfer of the tonal material is then much faster than in previous recording	Weighting increased; test how much this affects the adaptation Note V.: F. plays diatonic, S. first chromatic, but then immediately adopted by F. a lot, especially arpeggios

Table A.3.: Experimental validation table, Part III.

Date/Time	**Nr**	**Instrument**	**Technical remarks**	**Improv approach**	**Impression**	**Comments**
2021-04-15	5	MIDI-Keyboard	=	Mixed tonal material (free)	Result is somewhat confused; relatively much dissonance	Continue playing with unchanged parameters and now free tonal material Note V.: S. leaves a lot of pauses & is often a bit behind; did not adapt so quickly
2021-04-15	6	MIDI-Keyboard	<> Note-Type Intervals			
2021-04-15	7	MIDI-Keyboard	= Weighting 90%, DIY	Only fourths structures	No clear result to what extent Spirio has adopted the interval structure	Weighting strong for fastest possible adaption Experiment how Note-Type "Intervals" influence Spirio's response
2021-04-15	8	MIDI-Keyboard	=	Only fourths structures	More and more fourth structures are adopted, i.e. Spirio actually adopts interval structure	Previous recording not clear to what extent played structure is taken over (probably because of still existing pretraining) Therefore played on with unchanged parameters and same tonal material Note V.: S. takes over structure of fourths only after a long time and also not so extreme
2021-04-15	9	MIDI-Keyboard	=	Finishing pure fourth playing, free tonal material		Checking whether and how fast Spirio follows the tonal material Note V.: S. quickly forgot fourth structures
2021-04-15	10	MIDI-Keyboard	<> Layout:Joint	Diatonic (one key)	Frequent note repetitions; follows less in the key	Experiment whether it rather follows a key or rather takes over interval structures Note V.: S. rather does not adopt the key

Table A.4.: Experimental validation table, Part IV.

Date/Time	**Nr**	**Instrument**	**Technical remarks**	**Improv approach**	**Impression**	**Comments**
2021-04-15	11	Baritone saxophone	<> Note-Type Midikeys Layout Joint			
2021-04-15	12	=	= Weighting 90	F-maj		
2021-04-15	13	=	= Weighting 10/90?	Off key		Expectation: Spirio remains in previous key
2021-04-15	14	=	= Weighting 10		S. does not really go into what V. plays	Idea: Retraining parameters remotely controllable from Max patch; output parameter set via OSC
2021-04-15	15	=	<> Retraining Normal Sample-Rate 4, Nr Samples 15		S. adopts diatonics and is much more active	Expectation: Longer overflow, pauses are filled in Interaction meshes better, the two voices (Sax + Spirio) can develop better
2021-04-15	16	=	= Train-Rate 6, Sample-Rate 6, Nr Samples 20		Everything is very jumbled (from both sides)	
2021-04-15	17	=	= Nr Samples 25		Scale sounded great; very hip key change	
2021-04-15	18	=	=	More rests	S. has played many atonal lines	
2021-05-06	0	Alto saxophone	HMM			Generation vs Accompaniment Length Variability Content Variability Expressiveness Originality Interactivity Adaptability Control
2021-05-06	1	Alto saxophone	=			

A.4. Jazz Piano Chords

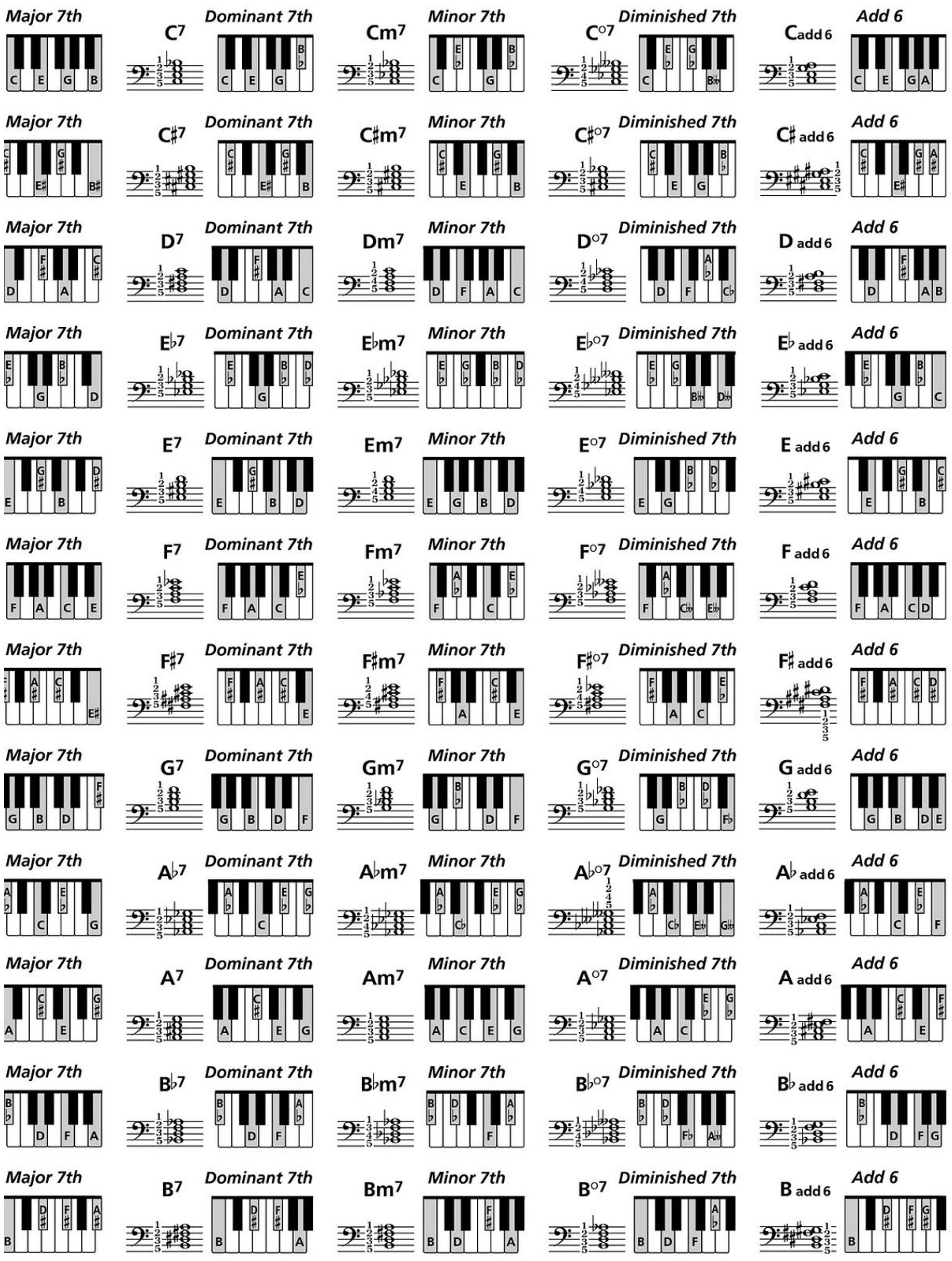


Figure A.2.: Overview of common jazz piano chords from [arxi 20].

List of Figures

List of Tables

List of Listings

Bibliography

[arxi 20] arxiusarquitectura. “How To Play Keyboard Chords”. Nov 2020. `https://www.arxiusarquitectura.cat/how-to-play-keyboard-chords/`, visited 2021-06-18.

[Baum 21] I. Baumann. *Prototypical development of an interactive, dynamic AI system for music generation and accompaniment based on existing datasets.* Master’s thesis, Technische Hochschule Nürnberg Georg Simon Ohm, Nuremberg, Apr 2021.

[Baum 72] L. E. Baum. “An inequality and associated maximization technique in statistical estimation for probabilistic functions of Markov processes”. In: O. Shisha, Ed., *Inequalities III: Proceedings of the 3rd Symposium on Inequalities*, Academic Press, 1972.

[Bech 15] J. Bechhoefer. “Hidden Markov models for stochastic thermodynamics”. *New Journal of Physics*, Vol. 17, No. 7, p. 075003, jul 2015.

[Bene 11] E. Benetos and S. Dixon. “A temporally-constrained convolutive probabilistic model for pitch detection”. 10 2011.

[Bene 16] D. Beney. “Useful MIDI Note Chart For Arduino Programming”. May 2016. Western Michigan University, `https://diymidicontroller.com/midi-note-chart/`, visited 2021-06-27.

[Berg 18] R. E. Berg. “Sound recording”. *Encyclopedia Britannica*, Jun 2018. `https://www.britannica.com/technology/sound-recording,visited2021-06-09`.

[Bhar 04] R. Bhar and S. Hamori. *Hidden Markov Models: Applications to Financial Economics.* Vol. 40 of *Advanced Studies in Theoretical and Applied Econometrics*, Springer US, 2004.

[Bjor] O. M. Bjørndalen. “MIDI Files”. Mido Documentation (v1.2.10), `https://mido.readthedocs.io/en/latest/midi_files.html`, visited 2021-06-02.

[Brin 20] S. Bringsjord and N. S. Govindarajulu. “Artificial Intelligence”. In: E. N. Zalta, Ed., *The Stanford Encyclopedia of Philosophy*, Metaphysics Research Lab, Stanford University, 2020.

[Brit 12] Britannica, The Editors of Encyclopaedia. “Improvisation”. *Encyclopedia Britannica*, Mar 2012. `https://www.britannica.com/art/improvisation-music`, visited 2021-05-27.

[Brit 18] Britannica, The Editors of Encyclopaedia. “Sampling”. *Encyclopedia Britannica*, Dec 2018. `https://www.britannica.com/science/sampling-statistics,visited2021-06-09`.

[Cote 21] J.-P. Côté. “WebMidi.js (v2.5.2)”. Feb 2021. GitHub, `https://github.com/djipco/webmidi`, visited 2021-05-27.

[Cycl 20] Cycling ’74. “Max/MSP: What ist Max?”. 2020. `https://cycling74.com/products/max`, visited 2021-05-18.

[Demp 77] A. P. Dempster and N. M. Laird. “Maximum likelihood from incomplete data via the EM algorithm”. *Journal of the Royal Statistical Society*, Vol. 39, No. 1, pp. 1–21, 1977.

[Fiel 00] R. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, Sep 2000. `http://roy.gbiv.com/pubs/dissertation/top.htm`, visited 2021-05-26.

[Goro 03] R. Gorow. *Hearing and Writing Music: Professional Training for Today’s Musician*. Gardena, CA: September Publishing, 2nd Ed., 2003.

[Grin 18a] M. Grinberg. “Flask documentation v2.0.x”. 2018. `https://flask.palletsprojects.com/en/2.0.x/`, visited 2021-05-27.

[Grin 18b] M. Grinberg. “Flask-SocketIO documentation”. 2018. `https://flask-socketio.readthedocs.io/en/latest/index.html`, visited 2021-05-27.

[Hida 17] Á. S. Hidalgo. “Generation of jazz improvisations in MATLAB”. July 2017. Universidad Politécnica de Madrid, `http://oa.upm.es/48942/`, visited 2021-06-09.

[Jura 09] D. Jurafsky and J. H. Martin. *Speech and language processing : an introduction to natural language processing, computational linguistics, and speech recognition*, Chap. Appendix A Hidden Markov Models. Pearson Prentice Hall, Upper Saddle River, N.J., 2009.

[Kath 15] T. Kathiresan. *Automatic Melody Generation.* Master's thesis, School of Electrical Engineering, KTH Royal Institute of Technology and Fraunhofer IDMT, Sweden, June 2015.

[Kohl 07] C. Kohlschein. "An introduction to hidden Markov models". Feb 2007. RWTH Aachen University.

[Kok 09] J. Kok. *ARTIFICIAL INTELLIGENCE*, Chap. Definition of Artificial Intelligenz. *Encyclopedia of life support systems*, Eolss Publishers Company, 2009.

[Lee 21] A. Lee. "hmmlearn: API Reference (v0.2.4)". Feb 2021. hmmlearn, `https://hmmlearn.readthedocs.io/en/latest/`, visited 2021-06-18.

[Lin 16] A. Lin. "Generating Music Using Markov Chains". Nov 2016. hackernoon, `https://hackernoon.com/generating-music-using-markov-chains-40c3f3f46405`, visited 2021-06-01.

[Liu 03] Q. Liu, Y.-S. Zhu, B.-H. Wang, and Y.-X. Li. "A HMM-based method to predict the transmembrane regions of β-barrel membrane proteins". *Computational Biology and Chemistry*, Vol. 27, No. 1, pp. 69–76, 2003.

[Mann 16] Y. Mann. "A.I. Duet". 2016. Google, `https://experiments.withgoogle.com/ai-duet`, visited 2021-05-27.

[Maro 97] Y. Marom. "Improvising Jazz With Markov Chains". Report as partial fulfilment of the requirements for the Honours Program of the Department of Computer Science, University of Western Australia, 1997.

[Meln 11] A. Melnikov and I. Fette. "The WebSocket Protocol". RFC 6455, Dec 2011. `https://rfc-editor.org/rfc/rfc6455.txt`, visited 2021-05-26.

[Niel 99] H. Nielsen, J. Mogul, L. M. Masinter, R. T. Fielding, J. Gettys, P. J. Leach, and T. Berners-Lee. "Hypertext Transfer Protocol – HTTP/1.1". RFC 2616, Jun 1999. `https://rfc-editor.org/rfc/rfc2616.txt`, visited 2021-05-26.

[Norg 13] M. Norgaard, M. Montiel, and J. Spencer. "Chords not required : Incorporating horizontal and vertical aspects independently in a computer improvisation algorithm". pp. 725–730, Proceedings of the International Symposium on Performance Science, 2013.

[npm 21] npm, Inc. "events v3.3.0". 2021. `https://www.npmjs.com/package/events`, visited 2021-05-27.

[Pian 21] Piano-Keyboard-Guide.com. "The A Minor Scale". 2021. `https://www.piano-keyboard-guide.com/a-minor-scale.html`.

[Post 80] J. Postel. “User Datagram Protocol”. RFC 768, Aug 1980. https://rfc-editor.org/rfc/rfc768.txt, visited 2021-05-26.

[Pree 20] Preetipadma. “CONTINUAL LEARNING: AN OVERVIEW INTO THE NEXT STAGE OF AI”. *Analytics Insight*, Oct 2020. https://www.analyticsinsight.net/continual-learning-an-overview-into-the-next-stage-of-ai/, visited 2021-06-11.

[Pyth 21] Python Software Foundation. “pickle: Python object serialization”. 2021. https://docs.python.org/3/library/pickle.html, visited 2021-05-27.

[Rabi 86] L. R. Rabiner and B. H. Juang. “An introduction to hidden Markov models”. *IEEE ASSp Magazine*, 1986.

[Raff 14] C. Raffel and D. P. W. Ellis. “Intuitive Analysis, Creation and Manipulation of MIDI Data with pretty_midi”. In: *15th International Conference on Music Information Retrieval Late Breaking and Demo Papers*, 2014.

[Rauc 21] G. Rauch. “SOCKET.IO v4”. Mar 2021. https://socket.io/, visited 2021-05-27.

[Rey 81] M. del Rey. “Transmission Control Protocol”. RFC 793, Sep 1981. https://rfc-editor.org/rfc/rfc793.txt, visited 2021-05-26.

[Sai 18] Sai. “Jazz ML ready MIDI”. 2018. https://www.kaggle.com/saikayala/jazz-ml-ready-midi, visited 2021-06-18.

[Schl 21] G. Schlag and B. Wenz. “Wenn Computer komponieren: Können Maschinen kreativ sein?”. Feb 2021. SWR2, https://www.swr.de/swr2/wissen/wenn-computer-komponieren-koennen-maschinen-kreativ-sein-swr2-wissen-2021-02-05-100.html, visited 2021-04-21.

[Schm 13] C. Schmidt-Jones. *Understanding Basic Music Theory.* Open textbook library. OpenStax CNX, 2013.

[Schu 95] E. Schukat-Talamazzini. *Automatische Spracherkennung: Grundlagen, statistische Modelle und effiziente Algorithmen. Künstliche Intelligenz*, W. Bibel and W. von Hahn, Vieweg Verlag, Braunschweig/Wiesbaden, 1995.

[Simo 08] I. Simon, D. Morris, and S. Basu. “MySong: Automatic Accompaniment Generation for Vocal Melodies”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, p. 725–734, Association for Computing Machinery, New York, NY, USA, 2008.

[Smit 03] S. Smith and Trident Press International. *The New International Webster's Comprehensive Dictionary of the English Language.* Trident Press International, 2003.

[Souz 10] C. Souza. "Hidden Markov Models in C#". Mar 2010. `http://crsouza.com/2010/03/23/hidden-markov-models-in-c/`, visited 2021-06-02.

[Star 95] T. Starner and A. Pentland. *Real-Time American Sign Language Visual Recognition From Video Using Hidden Markov Models.* Master's thesis, Massachusetts Institute of Technology, Cambridge, Feb 1995.

[Swif 97] A. Swift. "A brief Introduction to MIDI". May 1997. Imperial College of Science Technology and Medicine.

[The 21] The MIDI Association. "GM 1 Sound Set". 2021. `https://www.midi.org/specifications-old/item/gm-level-1-sound-set`, visited 2021-06-18.

[Trum 21] S. Trump, I. Agchar, I. Baumann, F. Braun, K. Riedhammer, L. Siemandel, and M. Ullrich. "Spirio Sessions: Experiments in Human-Machine Improvisation with a Digital Player Piano". In: *Proceedings of the 2nd Joint Conference on AI Music Creativity*, Graz, 2021.

[Vand 12] D. Vandenneucker. "MIDI Tutorial". 2012. Arpege Music, `http://www.music-software-development.com/midi-tutorial.html`, visited 2021-06-02.

[Vase 01] S. Vaseghi. *Hidden Markov Models*, Chap. 5, pp. 143–177. John Wiley & Sons, Ltd, 2001.

[West 14] Western Michigan University. "The Elements of Music". Jan 2014. Western Michigan University, `https://wmich.edu/mus-gened/mus150/Ch1-elements.pdf`, visited 2021-06-01.

[Wils 15] C. Wilson and J. Kalliokoski. "Web MIDI API: W3C Working Draft 17 March 2015". Mar 2015. World Wide Web Consortium (MIT, ERCIM, Keio, Beihang), `https://www.w3.org/TR/webmidi/`, visited 2021-05-27.

[Wrig 03] M. Wright, A. Freed, and A. Momeni. "OpenSound Control: State of the Art 2003". *Proceedings of the 2003 Conference on New Interfaces for Musical Expression (NIME-03)*, Jun 2003.

Glossary

API Application Programming Interface. i, 48

BPM beats per minute. i

CD Categorical Distribution. i

HMM Hidden Markov Model. i

HTTP Hypertext Transfer Protocol. i

JSON JavaScript Object Notation. i

MIDI Musical Instrument Digital Interface. i

MLE Maximum Likelihood Estimation. i

MM Markov Model. i

MSP Max Signal Processing. i

OSC Open Sound Control. i

SPN Scientific Pitch Notation. i

TCP Transmission Control Protocol. i

UI User Interface. i

User Datagram Protocol User Datagram Protocol. i

W3C World Wide Web Consortium. i